MASTER EN SCIENCES INFORMATIQUES

Génération de carrés magiques pour la génétique

Leblanc, Jean-Noël

Award date:
2007

Awarding institution:
Universite de Namur

FACULTES UNIVERSITAIRES
NOTRE-DAME DE LA PAIX
NAMUR

Institut d'informatique

Année académique 2006-2007

GENERATION DE CARRES MAGIQUES POUR LA GENETIQUE

Jean-Noël Leblanc

Mémoire présenté en vue de l'obtention du grade de maître en informatique.

Résumé

Ce travail consiste à étudier une méthode permettant de générer des carrés magiques réguliers additifs simples d'ordre compris entre 7 et 23 et dont les diagonales seront imposées. Après avoir défini différentes notions, une brève histoire des carrés magiques est présentée, ainsi que leur utilisation et les méthodes permettant d'en construire. Cela est suivi par la définition complète des objectifs de ce travail.

La seconde partie est consacrée aux méta-heuristiques et à l'étude des possibilités qu'elles offrent pour résoudre le problème présenté. Trois de ces méthodes sont étudiées, à savoir : le recuit simulé, la recherche avec tabous et la recherche adaptative. Pour chacune d'entre-elles, le fonctionnement, les avantages et inconvénients sont présentés ainsi que la façon dont la méthode peut être utilisée pour la génération de carrés magiques.

Les troisième et quatrième chapitres présentent l'implémentation complète d'une de ces méthodes : la recherche adaptative. L'algorithme complet est présenté et commenté, et les différents paramètres permettant de conduire la recherche sont étudiés.

Ce travail est conclu par un commentaire sur les résultats obtenus par la méthode élaborée et par plusieurs pistes permettant de prolonger le travail effectué.

Mots clés : Carré magique, méta-heuristique, recuit simulé, recherche avec tabous, recherche adaptative.

Abstract

This work consists in studying a method allowing the generation of simple additive regular magic squares with an order ranging from 7 to 23 and with imposed diagonals. After the definition of different notions, a brief story of magic squares is presented, as well as their use and methods allowing their constructions. That is followed of a complete definition of this work's objectives.

The second part is dedicated to meta-heuristics and a study of possibilities they offer in order to solve the presented problem. Three of them are studied, namely the simulated annealing, the Tabu search and the adaptive search. For each of them, the functioning, advantages and disadvantages are presented as well as the way in which the method can be used for the generation of magic squares.

The third and fourth parts present the complete implementation of one of these methods : the adaptive search. The complete algorithm is presented and explained, and the different parameters allowing to undertake the research are studied.

This work is concluded with observations on results obtained with the method developed and with several working tracks allowing the extension of the work performed.

Key words : magic square, meta-heuristic, simulated annealing, Tabu search, adaptive search

Avant-propos

Je tiens tout d'abord à remercier le professeur J. Fichefet, promoteur de ce mémoire, il a su me guider et m'épauler tout au long de ce travail. Je le remercie également pour le temps qu'il m'a consacré et pour les nombreuses remarques et commentaires qui m'ont aidés à finaliser ce mémoire.

Mes remerciements vont également au Docteur Jacques, co-promoteur de ce travail, il a su me faire découvrir les carrés magiques et une partie de leur mystère, qui m'a « inoculé le virus » comme il se plaît à le dire et qui m'a apporté de nombreuses précisions qui m'ont permis de bien cerner le sujet dès le début. Je tiens également à le remercier pour la confiance qu'il m'a donnée en me confiant les nombreux manuscrits qu'il a produits sur ce sujet.

Enfin, je remercie l'ensemble des personnes qui m'ont apporté, de près ou de loin, leur aide à l'étude, à l'élaboration et à la correction de ce travail.

Table des matières

Index des figures

Index des tableaux

Glossaire

Carré magique additif simple

Un carré magique additif simple est une grille carrée de nombres entiers dont la somme des termes suivant les lignes, les colonnes et les diagonales principales est la même.

Ordre d'un carré magique

L'ordre n d'un carré magique est le nombre de cases par côté de la grille carré le composant. C'est aussi la racine carrée du nombre de cases de cette grille.

Carré magique régulier

Un carré magique est dit régulier, ou normal, lorsque ses termes sont constitués par la suite naturelle des nombres entiers de 1 à n^2.

Constante magique

La constante magique, ou encore la somme magique, est la somme linéaire des nombres des lignes, des colonnes et des diagonales principales d'un carré magique additif simple. La constante magique k d'un carré additif simple régulier d'ordre n est d'équation $k = n * (n^2 + 1) / 2$

Carré bimagique

Un carré magique est dit « bimagique », ou encore, « satanique », lorsque le carré résultant de l'élévation au carré de chacun de ses termes est lui-même magique.

Métaheuristique

Les métaheuristiques forment une famille d'algorithmes d'optimisation visant à résoudre des problèmes d'optimisation difficile (souvent issus des domaines de la recherche opérationnelle, de l'ingénierie ou de l'intelligence artificielle) pour lesquels on ne connaît pas de méthode classique plus efficace.

INTRODUCTION

Depuis plusieurs siècles avant Jésus-Christ, les carrés magiques fascinent. Ils sont déjà connus dans la Chine antique où il leur était prêté des propriétés divines. En Europe, au Moyen-âge, ils apparaissent à de nombreuses reprises dans les grimoires des sorciers et magiciens. C'est également à cette époque que les alchimistes vont les utiliser pour modéliser les planètes ou certains métaux. A l'époque déjà, les carrés magiques étaient considérés comme de bons modèles de la nature.

C'est via les mathématiciens arabes que les carrés magiques arrivent en Europe comme objets d'études mathématiques. Ils seront étudiés par de grands mathématiciens et savants au cours des siècles. Relevons notamment que Pascal, Fermat, Euler, Benjamin Franklin parmi bien d'autres en ont étudié les caractéristiques et les méthodes de construction. A l'heure actuelle, on dénombre plus d'une centaine de méthodes permettant de construire de nombreux carrés magiques de tout ordre.

Au début du XX^{ème} siècle, Sir Ronald Fischer, de l'université de Cambridge, va utiliser les carrés magiques, alors essentiellement considérés comme une récréation mathématique, pour ses expériences statistiques lors de ses études sur les plantes. Il va, pour la mise au point de ses nouvelles méthodes d'expérimentation, prouver que l'utilisation de carrés latins et gréco-latins, cousins proches des carrés magiques, permettent de définir un plan d'expérimentation permettant de réduire de façon significative le nombre d'expériences tout en permettant d'en déduire un maximum d'informations. Celui qui est toujours considéré à l'heure actuelle comme le « *père des statistiques modernes* » va, à nouveau, utiliser les carrés magiques pour modéliser les phénomènes naturels.

Aujourd'hui, des chercheurs travaillant dans le domaine de la génétique tentent d'utiliser les carrés magiques pour modéliser la nature, et plus précisément, les supports de l'hérédité. Depuis longtemps, on sait que les chromosomes sont un support permettant d'expliquer l'hérédité mais, de récentes études tendent à montrer qu'ils ne permettent pas de l'expliquer à eux seuls. Ces études mettent en avant un autre support, extra-cellulaire celui-là, et dépendant de l'hérédité maternelle. Une des pistes retenues pour tenter de comprendre et d'expliquer l'influence de l'hérédité maternelle sur l'hérédité consiste à utiliser les carrés magiques en modélisant les chromosomes, ou une partie de ceux-ci, sur les diagonales du carré et en utilisant les autres nombres pour modéliser l'influence de l'hérédité maternelle.

Dans le présent travail, l'élaboration d'un outil destiné aux chercheurs travaillant dans cette voie est étudiée. L'objectif est de pouvoir générer un carré magique d'un ordre donné, lequel doit être construit autour de diagonales définies par l'utilisateur. Les carrés magiques générés devront en outre répondre à des règles précises propres à cette modélisation.

Le premier chapitre de ce travail est consacré aux carrés magiques, ils y sont définis et caractérisés précisément. Y sont présentées leur histoire et leur utilisation au fil des siècles. Ce chapitre permet également de d'aborder de façon plus détaillée le sujet de ce travail. Il se termine par l'analyse de quelques méthodes permettant de générer des carrés magiques et de l'examen de leurs limites. Il débouche enfin sur l'intérêt de rechercher un nouvel outil pour créer des carrés magiques correspondant à la modélisation des mécanismes de l'hérédité.

Le second chapitre est consacré à l'explication théorique de différentes méthodes devant permettre à un ordinateur de créer les carrés magiques demandés. La première méthode présentée est sans doute la plus intuitive, mais elle ne peut fonctionner que pour des carrés magiques ayant un ordre de grandeur très petit. Sont ensuite explorés trois

méthodes devant permettre de générer des carrés magiques de n'importe quel ordre. Ces méthodes sont toutes des méta-heuristiques, elles se basent donc sur des recherches aléatoires mais guidées, lesquelles sont particulièrement adaptées aux problèmes fortement combinatoires tel celui qui est étudié dans ce travail. Cette partie explique le principe général de chacune de ces méthodes, son fonctionnement, ses avantages et inconvénients ainsi qu'une étude montrant que l'outil recherché est réalisable via ces méthodes.

Le troisième chapitre traite de manière détaillée de l'algorithme implémenté suivant une des méthodes présentées au chapitre précédent : la recherche adaptative. Il s'agit de présenter l'algorithme général de la recherche avant de détailler chacune de ses parties. Pour chaque partie, sont présentés le principe de fonctionnement, l'algorithme et ses éventuelles subtilités techniques. Sont également abordées la complexité théorique de chacune des parties de l'algorithme ainsi que la discussion de certains paramètres ou choix d'implémentation.

Le quatrième chapitre, quant à lui, est consacré à l'explication et au dimensionnement des différents paramètres conduisant la recherche. Dans un premier temps, il reprend l'explication de l'influence théorique de ces différents paramètres sur le déroulement et les résultats de la recherche. Ensuite, il permet l'analyse et l'interprétation des différents jeux de tests effectués. De ces différents jeux de tests, seront déduites des valeurs de paramètres permettant un bon fonctionnement de la recherche, tant en terme de temps de réponse qu'en terme de taux de réussite.

Les conclusions tirent un bilan sur la qualité de l'outil créé, en terme de taux de réussite ainsi qu'en terme de performances. Elles présentent également quelques suggestions d'amélioration de l'outil existant, mais aussi de possibles recherches permettant de compléter ce qui a été étudié dans ce travail.

Chapitre I
LES CARRÉS MAGIQUES

I.1 Définition d'un carré magique

Un **carré magique additif simple régulier** est une grille contenant les n^2 premiers nombres, chacun occupant une case particulière. Ils y figurent chacun une et une seule fois, sans répétition. De plus, les sommes linéaires, prélevées suivant les lignes, les colonnes et les deux principales diagonales affichent toutes le même résultat. Ce résultat est appelé **constante magique**.

L'**ordre** d'un carré magique correspond au nombre de cases par côté de la grille, soit n pour un carré contenant n^2 cases.

La constante magique est la somme linéaire de chaque ligne, colonne et diagonale principale d'un carré magique. Elle se calcule facilement. En effet, la somme de tous les éléments du carré est la somme des n^2 premiers nombres entiers, soit $S = n^2 (n^2 + 1) / 2$. La somme constante à toutes les rangées est donc S / n et donc, $\boldsymbol{k = n * (n^2 + 1) / 2}$ ou k est la constante magique.

Voici un exemple d'un carré magique d'ordre 7 :

35	29	34	14	36	10	17
19	33	13	40	05	28	37
08	11	45	18	47	21	25
30	04	43	01	20	46	31
39	09	02	49	16	22	38
12	48	15	27	07	42	24
32	41	23	26	44	06	03

Figure I-1 : Exemple de carré magique additif simple normal

Les 49 premiers nombres s'y trouvent bien une et une seule fois, ce qui donne au carré son caractère **normal**. On constate également que les sommes linéaires de chaque ligne, colonne et diagonale principale valent 175, ce qui correspond à la constante magique de l'ordre 7. Cette dernière caractéristique permet de le qualifier de carré magique additif simple.

Il existe d'autres types de carrés magiques ainsi, par exemple, le **carré magique multiplicatif** lequel est un carré dont la constante magique est obtenue par multiplication successive des termes des lignes, colonnes et diagonales. Néanmoins, ce travail ne s'intéresse qu'aux **carrés magiques additifs simples normaux** que, par facilité, nous appellerons carrés magiques.

Un carré est dit bi-magique si, en plus d'être magique, la somme de chaque nombre préalablement élevé au carré, la somme sur les lignes, colonnes et diagonales est constante.

La constante bi-magique est également simple à calculer : elle est égale à la somme des n^2 premiers nombres préalablement élevés au carré, le tout divisé par n. Ce qui nous donne $\boldsymbol{k_2 = n * (n^2 + 1) * (2 * n^2 + 1) / 6}$ ou k_2 est la constante bi-magique.

Voici un exemple d'un carré bi-magique d'ordre 8 :

24	34	26	48	03	53	13	59
05	51	11	61	18	40	32	42
49	07	63	09	38	20	44	30
36	22	46	28	55	01	57	15
62	12	52	06	41	31	39	17
47	25	33	23	60	14	54	04
27	45	21	35	16	58	02	56
10	64	08	50	29	43	19	37

Figure I-2 : Exemple de carré bi-magique

Les 64 premiers nombres s'y retrouvent bien une et une seule fois. La somme linéaire de chaque ligne, de chaque colonne et des deux principales diagonales est de 260 et la somme de chaque nombre élevé au carré sur toutes les lignes, verticales et les deux diagonales principales est bien de 11180.

I.2 L'histoire des carrés magiques en quelques exemples

Les carrés magiques sont apparus très anciennement, notamment en Chine où le carré magique d'ordre 3, appelé Lo-Shu, est connu depuis les premiers siècles de notre ère. Il se présente sous la forme suivante :

4	9	2
3	5	7
8	1	6

Figure I-3 : Le Lo-Shu

Voici une légende de sa découverte, racontée par Philipp I.S.Lei : « *Aux temps anciens de la Chine, il y eut une grande inondation. Les gens tentèrent d'offrir quelque sacrifice au dieu de la rivière d'une des rivières en crue, la rivière Lo, pour calmer sa colère. Cependant, à chaque fois, une tortue sortait de la rivière et marchait autour d'eux. Le dieu de la rivière ignora leur sacrifice jusqu'à ce qu'une fois, un enfant remarqua la curieuse figure sur la carapace de la tortue. Alors les gens se rendirent compte que la valeur du sacrifice devait atteindre 15.* »

Le Lo-Shu peut être facilement résolu par un système d'équations. Il existe en réalité 8 carrés magiques d'ordre 3 qui se déduisent tous les uns des autres par symétrie ou rotation, le chiffre 5 se trouvant toujours dans la case centrale.

Le carré magique d'ordre 4, le plus ancien que nous connaissons, vient des Indes. Il a été retrouvé sur un des piliers d'un temple de Khajuraho, lequel date du X^{e} ou du XI^{e} siècle. Il est présenté ci-dessous :

07	12	01	14
02	13	08	11
16	03	10	05
09	06	15	04

Figure I-4 : Le carré de Khajuraho

Les carrés magiques sont déjà connus en Europe au Moyen-âge, notamment dans les grimoires des sorciers et magiciens. A cette époque, les alchimistes les associent aux métaux et aux planètes, et ce, en fonction de leur ordre. Ainsi, l'ordre 3 était associé à Saturne et au plomb, l'ordre 4 à Jupiter et à l'étain et ainsi de suite jusqu'à l'ordre 10 qui était associé à la Terre. Cette utilisation des carrés magiques est notamment explicitée par Henri-Corneille-Agrippa von Nettesheim dans son ouvrage *De Occulta Philosophia Libri III* (1533). On lui doit d'ailleurs 7 carrés magiques dont celui associé au Soleil (ordre 6) est présenté ci-après :

06	32	03	34	35	01
07	11	27	28	08	30
19	14	16	15	23	24
18	20	22	21	17	13
25	29	10	09	26	12
36	05	33	04	02	31

Figure I-5 : Le Soleil (selon Agrippa)

Après être passés entre les mains des mathématiciens arabes, c'est à la Renaissance que les carrés magiques font leur apparition en Europe Occidentale comme figure mathématique. Plusieurs personnages de l'époque vont s'y intéresser.

Un carré magique très célèbre de cette époque, d'ordre 4, est celui trouvé par Albrecht Dürer (1471-1528). Il l'a notamment rendu célèbre en le faisant figurer dans sa célèbre gravure « Die Melancolie ». Le voici :

16	03	02	13
05	10	11	08
09	06	07	12
04	15	14	01

Figure I-6 : Le carré de Dürer

Dans ce carré éminemment magique, on peut retrouver le nombre 34, c'est-à-dire la constante magique de l'ordre 4 de 24 manières différentes. Outre bien sûr les lignes, colonnes et diagonales, on notera que la somme des 4 cases centrales fait également 34, tout comme la somme des 4 cases angulaires. On retrouvera toujours cette même somme en divisant le carré en quartiers de 4 cases.

Un autre carré magique de l'époque, d'ordre 5 celui-là, est attribué au moine Manuel Moschopoulos :

10	18	01	14	22
04	12	25	08	16
23	06	19	02	15
17	05	13	21	09
11	24	07	20	03

Figure I-7 : Le carré de Moschopoulos

Outre son caractère magique, ce carré est également diabolique, ou panmagique, c'est-à-dire que la somme des éléments suivant ses diagonales parallèles est également de 65, la constante magique de l'ordre 5.

Les carrés magiques vont ensuite être étudiés par de nombreux mathématiciens, lesquels vont publier quantité d'ouvrages sur leurs caractéristiques et méthodes de construction. Parmi ceux-ci, notons de manière non exhaustive, Euler, Pascal, Fermat, ou encore, Benjamin Franklin.

C'est au début du XX^{ème} siècle que Fischer va les étudier à son tour et les utiliser dans sa nouvelle approche révolutionnaire de la statistique. A notre époque, les carrés magiques continuent de passionner. Il suffit, pour s'en rendre compte, de voir les nombreux ouvrages ou sites Internet en traitant.

Pour de plus amples informations sur les carrés magiques et leur histoire, on peut se référer à [Descombes, 2000a], ou encore [Bouteloup, 1991]

I.3 L'utilisation des carrés magiques

Les carrés magiques ont, par le passé, été utilisés comme talismans par les sorciers et magiciens qui s'en servaient pour représenter des éléments naturels comme des planètes ou des métaux. Ils ont également longtemps servi et servent toujours comme jeux récréatifs pour les férus de mathématiques et de magie des chiffres.

Néanmoins, ils ont également des utilisations beaucoup moins récréatives. Ainsi, Ronald Aymler Fischer (1890 – 1962), dont messieurs Droessbeke et Tassi diront qu'il est « *l'homme qui a fait de la statistique une science moderne* », va transformer ce que l'on considérait alors comme une récréation mathématique en procédure essentielle de la statistique.

En effet, alors que Ronald Fischer travaille sur l'étude des engrais, à la station agricole de Rothamsted, il est amené à conduire de nombreuses expériences. Cependant, compte tenu du temps qu'il faut pour mener une expérience à son terme (Il faut que la plante pousse) et la précision des données qu'il veut obtenir, il va essayer de réduire substantiellement le nombre d'expériences nécessaires. Et c'est en essayant de les organiser de façon à tirer un maximum d'informations de celles qui sont faites qu'il va inventer une méthode utilisant les carrés latins et gréco-latins, appelés également Eulériens en souvenir du célèbre mathématicien qui les a le premier étudiés. Sa méthode, publiée dans son ouvrage « *Statistical Methods for Research Workers* » (1925), est la suivante : supposons que l'on veut étudier un effet dépendant de 3 facteurs (x_1, x_2 et x_3) qui peuvent prendre 5 niveaux, ou valeurs, chacun.

- Une première méthode consiste à faire une expérience pour chaque valeur possible, ce qui donne 5^3 expériences, soit 125.
- La seconde méthode, celle de Fischer, consiste à déterminer quelles sont les combinaisons de valeurs des facteurs x_1, x_2 et x_3 qu'il faut expérimenter pour avoir le plus d'informations possibles. Fischer a montré qu'une bonne solution consiste à organiser les expériences selon un carré latin de 5x5 dans lequel chaque ligne correspond à une valeur de x_1, chaque colonne à une valeur de x_2 et le contenu de la case, à la valeur de x_3. Cela nous donne donc, par exemple, le carré latin suivant :

A	B	C	D	E
E	A	B	C	D
D	E	A	B	C
C	D	E	A	B
B	C	D	E	A

Figure I-8 : Carré latin correspondant un plan d'expérimentation de 3 facteurs à 5 valeurs

Si on veut ajouter à l'expérimentation un quatrième facteur x_4, lequel peut également prendre 5 valeurs, on utilisera un carré gréco-latin, ou Eulérien, pour déterminer le plan d'expérimentation. La première lettre (latine) de chaque case reprenant le niveau de x_3 et la seconde lettre (grecque) le niveau x_4. Dans un carré Eulérien, chaque couple de valeurs doit être unique. Voici le carré gréco-latin correspondant au plan d'expérimentation avec 4 facteurs pouvant prendre 5 niveaux chacun :

Aα	Bβ	Cγ	Dδ	Eε
Eβ	Aγ	Bδ	Cε	Dα
Dγ	Eδ	Aε	Bα	Cβ
Cδ	Dε	Eα	Aβ	Bγ
Bε	Cα	Dβ	Eγ	Aδ

Figure I-9 : Carré gréco-latin correspondant à un plan d'expérimentation de 4 facteurs à 5 valeurs

Pour plus d'informations sur l'histoire de la statistique et la contribution de Sir Fischer à cette science, on peut consulter [Droesbeke, 1997]

La méthode de Fischer s'applique aux carrés latins et eulériens mais pas aux carrés magiques proprement dits. Néanmoins, ces différents types de carrés sont intimement liés, de nombreuses méthodes de constructions de carrés magiques se basent d'ailleurs sur l'utilisation de tels carrés.

Notons enfin, à propos de Sir Fischer, que le vitrail du « Gonville and Caius College » à Cambridge dont une reproduction se trouve ci-dessous est construit autour d'un carré latin d'ordre 7 (le carré de Fischer) dans lequel le symbole utilisé est la couleur.

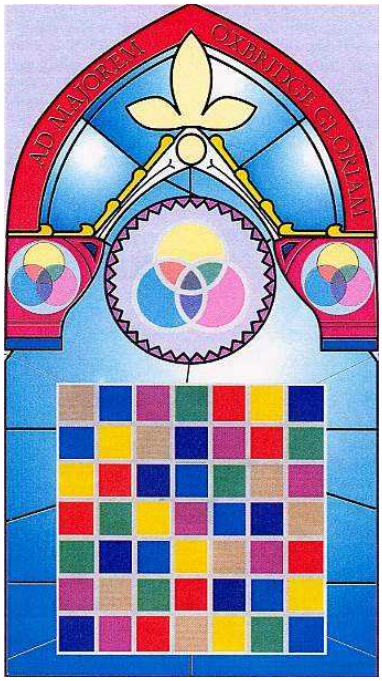


Figure I-10 : Le vitrail du "Gonville and Caius College" à Cambridge

I.4 Les carrés magiques et la génétique

De récentes études s'intéressant à l'hérédité maternelle ont amené plusieurs chercheurs à penser qu'elle ne peut pas s'expliquer uniquement par l'ADN. Ils estiment qu'il pourrait exister un autre support à l'hérédité, mais extra-nucléaire celui-là, et donc lié à la mère.

Comme nous l'avons vu au travers de leur histoire, depuis les alchimistes, en passant par de grands scientifiques comme Fischer, les carrés magiques ont été utilisés comme modèles analogiques de la nature.
Dans la même voie, l'idée est ici de tenter de modéliser les différents supports de l'hérédité en utilisant les carrés magiques. Les diagonales serviraient à modéliser un chromosome ou une partie de lui-même tandis que le reste des nombres servirait à explorer et expliquer le support extra-nucléaire, ou cytoplasmique, de l'hérédité.

Dans ce modèle, chacune des diagonales représente une des chromatides composant le chromosome, tandis que le noyau du carré, point d'intersection des diagonales, représentera le centromère du chromosome.

Le but de ce travail est de trouver une méthode permettant de générer un carré magique construit autour de diagonales fixées au départ. Il est bien entendu que lors de la construction du carré magique, il est nécessaire de tenir compte des contraintes propres au modèle.

Concrètement, cela signifie :

- Les éléments correspondant au centromère, c'est-à-dire les éléments se trouvant au centre du carré, doivent rester tels quels, il est interdit de les permuter.
- Les autres éléments se trouvant sur une des diagonales peuvent être changés de place pour autant qu'ils restent sur la même diagonale.
- Les autres éléments du carré peuvent être permutés entre eux mais ne peuvent être placés sur une des diagonales.

La taille du noyau du carré dépend de l'ordre n dans lequel on travaille. Si cet ordre est pair, le noyau se compose des 4 cases centrales du carré tandis que si l'ordre est impair, la seule case du noyau est la case centrale du carré.
Ci-dessous, deux exemples de problèmes posés et, pour chacun, une solution possible.

45						14
	25				44	
		16		21		
			15			
		35		20		
	12				17	
27						30

45	28	05	29	38	02	21
09	25	37	19	03	44	31
26	47	30	13	14	32	06
01	34	11	15	41	23	43
24	00	35	46	20	10	33
36	12	42	39	04	17	18
27	22	08	07	48	40	16

Figure I-11 : Un exemple du problème posé dans l'ordre 7 et une des solutions

63							62
	09					57	
		43			42		
			35	33			
			32	28			
		13			19		
	06					10	
07							45

63	56	01	23	17	25	05	62
26	45	29	51	04	03	57	37
12	34	43	11	58	42	52	00
47	21	08	35	33	39	53	16
41	30	48	32	28	20	15	38
02	46	13	55	22	19	36	59
54	06	49	18	40	44	10	31
07	14	61	27	50	60	24	09

Figure I-12 : Un exemple du problème posé dans l'ordre 8 et une des solutions

Pour chacun de ces carrés, les éléments repris en rouge sont ceux du noyau (ils sont fixés), les éléments en vert sont ceux des diagonales pouvant être permutés à l'intérieur de leur propre diagonale. Dans les solutions, on retrouve en bleu les éléments des diagonales ayant été permutés lors de la construction.

Il a été décidé de limiter la portée du travail aux carrés magiques d'ordre inférieur ou égal à 23.

I.5 Quelques méthodes de construction de carrés magiques

Il existe, à l'heure actuelle, plus d'une centaine de méthodes pour construire des carrés magiques. Ces méthodes permettent de trouver certains carrés magiques pour un ordre ou un groupe d'ordre donné. Le nombre de carrés générés par chaque méthode est très variable. Certaines sont très peu prolifiques alors que d'autres peuvent en générer de grandes quantités. Ces méthodes se basent sur une ou plusieurs caractéristiques de carrés magiques ou d'ordres particuliers. Elles se basent, pour la plupart, sur la combinaison de carrés latins, eux-mêmes créés selon un procédé strict.

I.5.1 La méthode de La Hire.

Cette méthode, issue des travaux de Philippe de La Hire (1640 - 1768), est également appelée « Méthode des horizontales ». Elle ne fonctionne que pour les carrés d'ordre n impair.

Elle se base sur la construction de deux carrés latins auxiliaires de même ordre que le carré magique à construire. Les carrés latins peuvent être définis comme suit : *« Dans un ensemble E de n éléments, on appelle « carré latin » d'ordre n, une grille de n² cases, dans laquelle toute ligne et toute colonne contient une fois et une fois seulement, tout élément de l'ensemble E donné. Ces éléments peuvent être de toutes natures (lettres, chiffres, …) »*

A	C	B
B	A	C
C	B	A

Figure I-13 : Un exemple de carré latin

Cette méthode utilise deux carrés latins, construits suivant une technique précise, pour générer des carrés magiques.

Le premier des deux carrés latins est construit de la manière suivante :
Dans la première ligne, la série des n premiers entiers est écrite dans un ordre quelconque. Dans la seconde ligne, on écrira la série dans le même ordre, mais en commençant par le nombre se trouvant dans la colonne de la ligne précédente qui suit directement la colonne centrale. On procédera de la même manière pour les lignes suivantes. Dans l'ordre 5, cela nous donne, par exemple, le carré suivant :

5	3	2	4	1
4	1	5	3	2
3	2	4	1	5
1	5	3	2	4
2	4	1	5	3

Figure I-14 : Méthode de La Hire, construction du carré latin A

Le second carré latin est construit en mettant, dans un ordre quelconque, les éléments de la suite 0, n, 2n, 3n,… dans la première ligne. Pour les autres lignes, la même liste sera écrite dans le même ordre, mais en commençant par l'élément se trouvant au centre de la ligne précédente. Cela donnera donc, par exemple, le carré latin suivant :

15	5	0	20	10
0	20	10	15	5
10	15	5	0	20
5	0	20	10	15
20	10	15	5	0

Figure I-15 : Méthode de La Hire, construction du carré latin B

Pour obtenir un carré magique, il reste à faire la somme des deux carrés créés. On obtient ainsi :

20	08	02	24	11
04	21	15	18	07
13	17	09	01	25
06	05	23	12	19
22	14	16	10	03

Figure I-16 : Un carré magique d'ordre 5 obtenu par la méthode de La Hire

Les sommes linéaires de toutes les lignes, colonnes et diagonales principales sont bien de 65.

Cette méthode ne permet pas de générer tous les carrés magiques d'ordre 5. En effet, il y a 5! choix possibles pour la première ligne du premier carré et 5! choix possibles pour la première ligne du second carré. Cette méthode permet donc de trouver 14400 carrés magiques d'ordre 5. De manière plus générale, elle permettra, pour un ordre n impair de trouver n! x n! carrés magiques.

Il existe plusieurs variantes de cette méthode qui consistent à modifier la façon dont les carrés latins auxiliaires sont créés. Pour plus de renseignements sur ces méthodes, on peut consulter [Descombes, 2000b]

I.5.2 La méthode de Ralph Strachey

Cette méthode, qui s'applique aux carrés d'ordre n = 2*(2k+1), est à la fois longue et assez complexe, mais elle permet de générer un nombre tout à fait impressionnant de carrés magiques. Pour notre exemple, nous travaillerons dans l'ordre 10.

Pour commencer, il faut diviser le carré en 4 sous-carrés (A,B,C et D) suivant les médianes de la grille originale.

Phase I : Remplir le sous-carré A

Il est rempli avec les chiffres 0 et 3 en respectant les deux règles suivantes :

- Il doit y voir deux chiffres 3 par ligne.
- La diagonale descendante (du haut à gauche vers le bas à droite) doit contenir 3 fois le chiffre 3.

3	0	0	3	0					
3	0	0	0	3			B		
0	0	A	3	0					
0	3	0	0	3					
0	0	3	0	3					
		C					D		

Figure I-17 : La méthode de Ralph Strachey - Phase I

Phase II : Remplir le sous-carré C

Ce sous-carré est complété en observant la valeur de l'élément se trouvant dans la case correspondante du sous-carré A en prenant comme axe de symétrie leur côté commun. Si la valeur de cet élément est 3, il faut mettre 0 et vice-versa.

3	0	0	3	0					
3	0	0	0	3			B		
0	0	A	3	0					
0	3	0	0	3					
0	0	3	0	3					
3	3	0	3	0					
3	0	3	3	0					
3	3	C	0	3			D		
0	3	3	3	0					
0	3	3	0	3					

Figure I-18 : La méthode de Ralph Strachey - Phase II

Dans l'exemple ci-dessus, l'axe de symétrie est repris en rouge et des exemples de cases correspondantes sont marqués dans des couleurs similaires.

Phase III : Remplir le sous-carré B

On le complète avec les chiffres 1 et 2 et selon les deux règles suivantes :

- Le chiffre 2 doit apparaître quatre fois par ligne
- La diagonale descendante (du haut à droite vers le bas à gauche) doit contenir quatre fois le chiffre 2.

Ce qui donne, par exemple, :

3	0	0	3	0	2	1	2	2	2
3	0	0	0	3	2	1	2	2	2
0	0	3	3	0	1	2	2	2	2
0	3	0	0	3	2	1	2	2	2
0	0	3	0	3	2	2	2	1	2
3	3	0	3	0					
3	0	3	3	0					
3	3	0	0	3					
0	3	3	3	0					
0	3	3	0	3					

A B C D

Figure I-19 : La méthode de Ralph Strachey - Phase III

Phase IV : Compléter le sous-carré D

Le sous-carré D est rempli par rapport à B de la même façon que le sous-carré C a été rempli par rapport à A. Cela donne donc :

3	0	0	3	0	2	1	2	2	2
3	0	0	0	3	2	1	2	2	2
0	0	3	3	0	1	2	2	2	2
0	3	0	0	3	2	1	2	2	2
0	0	3	0	3	2	2	2	1	2
3	3	0	3	0	1	1	1	2	1
3	0	3	3	0	1	2	1	1	1
3	3	0	0	3	2	1	1	1	1
0	3	3	3	0	1	2	1	1	1
0	3	3	0	3	1	2	1	1	1

A B C D

Figure I-20 : La méthode de Ralph Strachey - Phase IV

Phase V : Chaque élément de la grille globale est multiplié par $n^2/4$. (soit 25 dans notre cas)

75	0	0	75	0	50	25	50	50	50
75	0	0	0	75	50	25	50	50	50
0	0	75	75	0	25	50	50	50	50
0	75	0	0	75	50	25	50	50	50
0	0	75	0	75	50	50	50	25	50
75	75	0	75	0	25	25	25	50	25
75	0	75	75	0	25	50	25	25	25
75	75	0	0	75	50	25	25	25	25
0	75	75	75	0	25	50	25	25	25
0	75	75	0	75	25	50	25	25	25

Figure I-21 : La méthode de Ralph Strachey - Phase V

Notons que si ce carré n'est pas n'est pas encore magique additif simple et régulier, il est déjà magique. En effet, la somme linéaire selon ses lignes, colonnes et diagonales principales est déjà constante et vaut 375.

Phase VI : Ramener le carré trouvé à une forme normale

Pour cette phase, la méthode va utiliser un carré magique d'ordre 5, choisi au hasard parmi les nombreux carrés magiques de cet ordre. Ce carré sera ajouté aux carrés A et B tandis que c'est le symétrique de ce carré par rapport à la médiane horizontale qui sera ajouté aux carrés C et D.

Prenons, par exemple, comme carré d'ordre 5, le carré de présenté en Figure I-7, celui de Moschopoulos.

10	18	01	14	22
04	12	25	08	16
23	06	19	02	15
17	05	13	21	09
11	24	07	20	03

11	24	07	20	03
17	05	13	21	09
23	06	19	02	15
04	12	25	08	16
10	18	01	14	22

Figure I-22 : Le carré de Moschopoulos et son symétrique par rapport à la médiane horizontale

Le carré magique d'ordre 10 généré est:

85	18	01	89	22	60	43	51	64	72
79	12	25	08	91	54	37	75	58	66
23	06	94	77	15	48	56	69	52	65
17	80	13	21	84	67	30	63	71	59
11	24	82	20	78	61	74	57	45	53
86	99	07	95	03	36	49	32	70	28
92	05	88	96	09	42	55	38	46	34
98	81	19	02	90	73	31	44	27	40
04	87	100	83	16	29	62	50	33	41
10	93	76	14	97	35	68	26	39	47

Figure I-23 : La méthode de Ralph Strachey - Phase VI

Cette méthode permet de créer un très grand nombre de carrés magiques. Ainsi, selon les calculs de L. Gérardin [Gérardin, 1986], pour l'ordre 10, il y a 10 combinaisons possibles par ligne, ce qui donne 100000 possibilités de remplir les 5 lignes du sous-carré A. En ajoutant la contrainte sur la diagonale, ce nombre est diminué pour être ramené à 16360. De même, pour le sous-carré B, il y a 5 façons de remplir chaque ligne, ce qui mène à 3125 possibilités de remplissage. De nouveau, le nombre de possibilités est réduit, cela étant toujours dû à la contrainte sur la diagonale, il y en aura finalement 1280. Sachant que l'on compte 275305224 carrés magiques d'ordre 5 que l'on peut combiner avec les grilles générées, le nombre de carrés que l'on peut générer avec la méthode de Ralph Strachey s'élève à 15360 x 1280 x 275305224 = 5,4 x $10^{15,}$ soit 5 millions de milliards…

Cette méthode fonctionne aussi pour les carrés d'ordre pairement paire, c'est-à-dire les ordres multiples de 4. La seule différence réside dans la façon de remplir les sous-carrés A et B où chaque ligne et chaque diagonale doit comporter le même nombre de 0 ; 3 ou 1 ; 2. On utilisera bien entendu un carré d'ordre 4 pour la normalisation du carré obtenu.

Tous les détails de cette méthode sont repris très en détail dans [Descombes, 2000c]

I.5.3 Les méthodes existantes et l'informatique

Les méthodes présentées dans ce chapitre sont loin d'être les seules existantes. Pour un examens plus détaillé de différentes méthodes, on peut se référer aux ouvrages de René Descombes [Descombes, 2000a] ou Lucien Gérardin [Gérardin, 1986] qui en reprennent de nombreuses.

Toutes les méthodes existantes peuvent être transposées à l'informatique facilement. Il s'agit en effet d'appliquer une « recette » bien définie, ce qui peut-être fait de manière très rapide et bien plus sûre que manuellement par un petit algorithme.

Cependant, ces méthodes ne permettent pas de répondre au problème posé ici. En effet, ces méthodes permettent difficilement, par leur mode de construction, de fixer les diagonales. De plus, chaque méthode permet de construire un certain type de carrés magiques. Mais aucune ne permet de construire n'importe quel type de carrés magiques.

Chapitre II
L'ETUDE THEORIQUE DE DIFFERENTES METHODES DE RESOLUTION POSSIBLES

II.1 Résolution du problème par une méthode complète

II.1.1 Introduction

Après différentes lectures, dont de nombreux écrits du docteur Jacques, la première possibilité étudiée a été de résoudre le problème par un algorithme classique. L'idée, d'ailleurs émise par le docteur Jacques dans son fascicule [Jacques, 2006], consistait à générer toutes les lignes magiques, ou bi-magiques si l'on s'intéresse au cas de la magie d'ordre 2, pour ensuite les placer pour former des carrés magiques.

Cette partie présente la première idée développée. Ses avantages, ses inconvénients, ses limites et, pour finir, les raisons pour lesquelles elle a été abandonnée.

II.1.2 L'idée conductrice de la méthode

L'idée suivie était de décomposer le problème en deux grandes parties distinctes que l'on pourrait résoudre indépendamment. La première partie consistait à calculer l'ensemble des lignes magiques pour un ordre donné. La seconde partie visait à choisir des lignes parmi celles calculées et à les placer en lignes et en colonnes pour former un carré magique, et ce, en respectant les diagonales fixées par hypothèses.

II.1.2.1 Calcul de l'ensemble des lignes magiques de l'ordre n

Problème : Soit l'ordre n, on veut calculer l'ensemble des vecteurs $[x_1, \ldots, x_n]$ tels que

- $x_1 + x_2 + \ldots + x_n = S_n$ où S_n est la somme magique d'ordre n
- $x_1, \ldots, x_n \leq n^2$
- $x_1 \neq x_2 \neq \ldots \neq x_n$
- $x_1, \ldots, x_n \in N^0$, dans lequel N^0 est l'ensemble des entiers naturels non nuls

Hypothèse : Calculer les vecteurs tels que $x_1 > x_2 > \ldots > x_n$, sachant que tous les vecteurs résultant de permutations des $x_1, \ldots, x_n$ possèdent les mêmes propriétés.

Le problème se résout en bornant les valeurs possibles pour les différents x_i. Ainsi, pour x_1, la valeur maximum possible est n^2. Pour ce qui est de la valeur minimum, par définition, on sait que

- $x_1 + x_2 + \ldots + x_n = n * (n^2 + 1) / 2$ (2.1)
- $x_1 > x_2 > \ldots > x_n$ (2.2)

Pour que x_1 soit minimum, il faut donc que $x_2 + \ldots + x_n$ ait une valeur maximum dans l'équation 2.1. Ce qui donne

$x_1 + (x_1 - 1) + (x_1 - 2) + \ldots + (x_1 - (n - 1)) = S_n$

$\Leftrightarrow \quad n * x_1 + n * (n - 1) / 2 = S_n$

$\Leftrightarrow \quad x_1 = (S_n + (n * (n - 1) / 2)) / n$ (2.3)

Pour chaque valeur de x_1 comprise entre les bornes calculées, les bornes supérieures et inférieures de x_2 seront calculées.
L'équation 2.1 devient alors $x_2 + x_3 + \ldots + x_n = S_n - x_1$ (2.4)

Dès lors, pour respecter les équations 2.2 et 2.4, on peut définir la valeur maximum de x_2 comme la plus petite entre $S_n - x_1 - 1$ (à cause de 2.4) et $x_1 - 1$ (à cause de 2.2)

Pour ce qui est de sa valeur minimum, on pourra utiliser la formule 2.3 en remplaçant S_n par $S_n - x_1$. Cependant, dans ce cas, le résultat n'est pas toujours entier. Or comme $x_2 \in N^0$, on devra arrondir à la valeur supérieure.

On remarquera ici que le problème de résultat non-entier ne se pose pas pour le calcul de x_1 car S_n vaut $n * (n^2 + 1) / 2$. Or, si on injecte cette valeur dans l'équation 2.3, on arrive facilement, par simplification, à l'équation $x_1 = n * (n + 1) / 2$, laquelle donne toujours un résultat entier.

Pour chaque valeur de x_2 comprise entre les bornes calculées, la méthode sera répétée pour le calcul des valeurs possibles de x_3, et ainsi de suite jusque x_n de manière récursive.

Cette méthode est implémentée via un algorithme récursif relativement simple que voici.

Génération de lignes magiques

Soit n, l'ordre dont on veut générer les lignes magiques.

```
{
    sommeMagique = n * (n² + 1) / 2

    si n = 1 alors
        LigneMagique[1] = 1
    Sinon
    {
        x1max = n * n
        x1min = (sommeMagique + (( n – 1) * n / 2)) / n

        ur x1 = x1Max et tant que x1 ≥ x1min par pas de –1 sur x1
        {
              ligneMagique[1] = x1
              appel à fCalculIteratif (n – 1, sommeMagique – x1, x1, 1, n)
          }
    }
}
```

Algorithme de fCalculIteratif

Soit les paramètres suivants :

- ⇒ *nbTermesACalculer : Le nombre de termes qu'il faut encore calculer dans la ligne magique*
- ⇒ *somme : la somme que les éléments restant doivent atteindre*
- ⇒ *xMax : La valeur maximum qu'il est possible d'affecter pour les termes restant à calculer.*
- ⇒ *nbElementsCalcule : Nombre d'éléments de la ligne déjà calculés.*
- ⇒ *ordre : Ordre de la ligne magique que l'on veut générer.*

```
{
    Si nbTermesACalculer > 1 alors     // Il reste plusieurs éléments à rechercher
    {
        xMin = (somme +( (ordre - 1) * nbTermesACalculer/ 2)) / nbTermesACalculer
        Arrondir xMin à la valeur entière supérieure

        Si somme < xMax alors
            xiMax = somme – 1
        Sinon
            XiMax= xmax – 1

        Pour xi = xiMax et tant que xi ≥ xiMin par pas de –1 sur xi
        {
            ligneMagique[nbElementsCalcule  + 1] = xi
            appel à fCalculIteratif (nbTermesACalculer  – 1, somm – xi, xi, nbElementsCalcule + 1 , ordre)
        }
    }
    Sinon            // Il ne reste qu'un élément à chercher dans la ligne
    {
        ligneMagique[nbElementsCalcule  + 1] = somme
        Sauvegarder la ligne magique contenue dans ligneMagique
    }
}
```

Figure II-1 : Algorithme de génération des lignes magiques d'ordre n

La complexité de cet algorithme est de l'ordre théorique $O(n^n)$, même si ce dernier est pratiquement plus limité puisque chaque x_i est borné. L'ordre de grandeur est en fait exactement identique à celui du nombre de lignes magiques pour un ordre donné puisque, via cet algorithme, chaque ligne magique est parcourue une et une seule fois. Cet ordre de grandeur n'en reste pas moins exponentiel.

Pour ce qui est de la recherche de lignes magiques contenant des ordres de magie supérieurs, la solution trouvée est de vérifier lors de la génération de chaque ligne magique du premier ordre que la ligne trouvée ne possède pas les caractéristiques des ordres de magie supérieurs.

Un code source en C implémentant cet algorithme est fourni en annexe (.A.2) de ce travail. Il permet de générer les lignes magiques d'un ordre donné et de vérifier si les lignes possèdent également les deuxième et troisième degrés de magie. Les lignes générées sont sauvées dans un fichier.

II.1.2.2 Formation du carré magique à partir des lignes magiques calculées

Le problème est ici beaucoup plus complexe. En effet, même en raisonnant sur un petit ordre de grandeur, comme l'ordre 4 ou 5, le nombre de possibilités de choix de lignes, de positions possibles pour les lignes choisies et de permutations de nombres dans une même ligne est tout à fait impressionnant.

L'idée suivie était, dans un premier temps, de restreindre l'ensemble des lignes magiques désormais connues pour ne garder que celles pouvant correspondre avec les diagonales du carré, lesquelles étaient fixées par hypothèse. Dans un deuxième temps, le but était de positionner les lignes entre-elles en retenant les différents choix posés dans un arbre de recherche.

Plusieurs algorithmes de type « générer et tester » ont été développés mais les arbres de recherches sont énormes à traiter. De plus, le nombre d'instanciations peut être très grand avant de pouvoir valider ou pas les choix posés. Enfin, il est impossible de choisir un ordre d'instanciation permettant de modifier la rapidité de l'algorithme de manière sensible. Le résultat mène donc à des algorithmes extrêmement lents et inefficaces.

II.1.2.3 Les limites de cette méthode.

Comme nous l'avons vu précédemment, cette méthode a rapidement montré des limites qui ont provoqué son abandon.

Outre la lenteur de l'algorithme, les données ne peuvent être enregistrées pour chaque nœud de l'arbre. Il est en effet indispensable de retenir de très nombreuses données pour définir la configuration courante pour le mécanisme de backtracking.

De plus, l'utilisation des données calculées lors de la première partie, est finalement difficile à utiliser dans la seconde. En effet, le nombre de lignes magiques que l'on peut utiliser et les nombreuses permutations que l'on peut faire avec les éléments les composant ne réduisent pas suffisamment le domaine des variables que pour être réellement intéressants.

Enfin, les calculs et vérifications à faire lors de chaque instanciation sont complexes et demandent un parcours de toutes les variables déjà instanciées. Notamment pour vérifier l'unicité des valeurs instanciées.

En conclusion, il est apparu que mon expérience dans la programmation classique et l'idée ressortant de mes lectures m'ont envoyé sur une fausse piste. Des méthodes plus adaptées ont alors été recherchées.

II.2 Les méthodes incomplètes et les meta-heuristiques

II.2.1 Introduction

Comme exposé dans la partie précédente, utiliser une méthode permettant la recherche exhaustive de solutions par une méthode complète est impossible étant donné l'énormité du domaine de recherche. En effet, même si seuls les ordres de grandeurs les plus petits sont traités, ces méthodes sont impossibles à mettre en œuvre dès que l'ordre de grandeur augmente un tant soit peu.

Le problème posé est un problème d'optimisation combinatoire, c'est-à-dire un problème dans lequel la meilleure solution doit être trouvée dans un ensemble de solutions réalisables, cet ensemble de solutions étant fini mais comptant un nombre très important d'éléments. Il est décrit par un ensemble de contraintes que les solutions réalisables doivent satisfaire.

La modélisation d'un carré magique d'ordre n en un ensemble de contraintes peut être faite comme suit :

- Soit x[i, j] les n^2 éléments de ce carré avec i = 1, …, n et j = 1, …, n. i est l'indice de la ligne de x et j est l'indice de la colonne de x
- Soit L_1, .., L_n la somme des éléments des lignes 1 à n avec L_k = x[k, 1] + x[k, 2] + … + x[k, n-1] + x[k, n]
- Soit C_1, .., C_n la somme des éléments des lignes 1 à n avec C_k = x[1, k] + x[2, k] + … + x[n-1, k] + x[n, k]
- Soit D1, la somme des éléments de la $1^{ère}$ diagonale avec D1 = x[1, 1] + x[2, 2] + … + x[n-1, n-1] + x[n ,n]
- Soit D2, la somme des éléments de la $2^{ème}$ diagonale avec D2 = x[1, n] + x[2, n-1] + … + x[n-1, 2] + x[n, 1]

Pour avoir un carré magique régulier d'ordre n, il faut respecter les contraintes suivantes :

- L1 = L2 = … = Ln = C1 = C2 = … = Cn = D1 = D2 = S avec S qui est la constante magique d'ordre n.
- $\forall i \; \forall j$, x[i, j] $\leq n^2$ avec $1 \leq i \leq n$ et $1 \leq j \leq n$
- $\forall i \; \forall j \; \forall k \; \forall l$, x[i, j] $\neq$ x[k, l] avec $1 \leq i \leq n$,$1 \leq j \leq n$, $1 \leq k \leq n$ et $1 \leq l \leq n$
- $\forall i \; \forall j$, x[i, j] $\in N^0$, dans lequel N^0 est l'ensemble des entiers naturels non nul.

Pour résoudre ce genre de problème, des méta-heuristiques vont être utilisées, c'est-à-dire des algorithmes itératifs où l'on va progresser de manière plus ou moins stochastique vers un optimum d'une fonction-objectif. Ces méthodes permettent de résoudre une large gamme de problèmes grâce à leur haut niveau d'abstraction. Elles sont souvent inspirées par des systèmes naturels (physique, biologie, comportements sociaux,…)

Dans le cadre de ce travail, j'ai constaté qu'il existait un nombre impressionnant de ces meta-heuristiques, ainsi que des variantes possibles en hybridant ces méthodes, que ce soit entre elles ou avec d'autres types de méthodes.

Trois méthodes ont été approfondies car elles semblent répondre aux exigences de cette étude particulière : la méthode du recuit simulé, la recherche Tabou et la recherche adaptative.

II.2.2 La méthode du Recuit Simulé

Cette méta-heuristique, issue des travaux de Kirkpatrick – Gelatt - Vecchi (1982) et Cerny (1985), se base sur des notions de physique des matériaux, et plus précisément sur les principes thermodynamiques de refroidissement des aciers.

Le procédé industriel dont cette méthode s'inspire est celui de la fabrication d'alliages métalliques par lequel on leur assure une structure sans défaut en le laissant refroidir par paliers de température successifs.

II.2.2.1 Les éléments de thermodynamique utilisés

Les éléments développés ici sont largement tirés du cours d'optimisation combinatoire du Professeur Leclercq [Leclercq, 2005]. Ils ont été complétés par des renseignements issus de [Dréo et al, 2005a] et de [Aarts et al, 1990]

Commençons par définir quelques notions indispensables pour la suite :

- **La configuration (c) d'un système** est l'ensemble des positions des particules de ce système.
- **Pr(c,t)** est la probabilité que le système soit dans une configuration **c** au moment **t**.
- **L'énergie, E(c),** est une fonction qui ne dépend que de la configuration.
- **L'état d'équilibre statistique** est l'ensemble de configurations c_i telles que $Pr(c_i,t)$ est indépendante de t.

A l'état d'équilibre statique, la probabilité d'une configuration c ne dépend donc que de l'énergie et de la température. La relation suivante peut être considérée :

$Pr_T(c) = e^{-E(c)/kT}/Z_T$ où Z_T est une constante et k la constante de Boltzmann.

Pour étudier l'évolution du système, il faut étudier le rapport entre deux probabilités de configurations, ce qui donne :

$Pr(c') / Pr(c) = e^{-(\Delta E/KT)}$

Avec : ΔE est la variation d'énergie, c'est à dire E(c') – E(c)

Dans cette formule, on voit que la probabilité de transition depuis la configuration c vers c' est d'autant plus importante que ΔE est fortement négatif. Le système évoluera donc vers des configurations avec une énergie de plus en plus faible.

Pour ce qui est de l'effet de la température, si elle est élevée, la probabilité de transition tendra vers 1, et ce, même si le ΔE est positif. Des configurations moins intéressantes du point de vue énergétique pourront alors être atteintes. L'énergie va néanmoins finir par se stabiliser à un régime stationnaire. Si la température est basse, il sera quasi-impossible de passer à une configuration avec un niveau d'énergie plus élevé. On va donc tendre vers des configurations d'énergie minimum. Néanmoins, travailler directement à des températures basses n'est pas possible car la recherche se figerait dans des minima locaux.

Le processus suivi pour la fabrication des alliages consiste à chauffer fortement le système. Celui-ci va évoluer fortement mais va se stabiliser à des configurations de plus faible énergie. Le refroidissement sera alors progressif, avec une stabilisation à chaque palier. Et ce, jusqu'à obtenir un état d'équilibre avec les configurations d'énergie minimum.

II.2.2.2 L'algorithme du Recuit Simulé

L'adaptation de la méthode à l'optimisation combinatoire se fait en remplaçant l'énergie par une fonction-objectif, ou fonction de coût qu'il faudra optimiser, et en remplaçant la température et la constante de Boltzmann par un paramètre qui jouera le même rôle.

Pour exécuter cet algorithme, il est nécessaire d'utiliser 4 paramètres qu'il y a lieu de dimensionner avec le plus grand soin pour assurer le bon fonctionnement de l'optimisation. Ce sont les suivants:

- La température initiale (T0)
- La température finale (TF)
- Le facteur de décroissance de la température (DC)
- Le nombre maximum d'itérations par palier de température (NBIter)

L'algorithme général donne donc :

Soit T, la température courante et FO, la fonction-objectif à optimiser.

Initialisation des 4 paramètres.
Construction d'une configuration de départ aléatoire.
T := T0
Tant que ($T \geq TF$) et ($FO > 0$) faire
i :=1
tant que ($i \leq NBIter$) et ($FO > 0$) faire
Soit c, la configuration courante telle que $c = (x_1,\ldots,x_k,\ldots,x_n)$
Choix d'un élément au hasard, soit le $k^{ème}$.
Choix d'un nouvel état, soit x'_k, pour cet objet. On obtient ainsi une configuration c' voisine de la première.
On calcule $\Delta E = E(c') - E(c)$
Si $\Delta E \leq 0$
On remplace c par c'
Sinon
On calcule $R = e^{-(\Delta E/T)}$ et on génère s, un nombre aléatoire de [0,1]
Si $s \leq R$,
c reste la configuration courante
Sinon,
c est remplacée par c'
i := i + 1
Fin tant que
*T := T * DC*
Fin tant que

II.2.2.3 Les avantages et les inconvénients du recuit simulé

Un premier avantage important est sa facilité d'implémentation et de mise en œuvre. Comme vu précédemment, l'implémentation de l'algorithme ne pose pas de problèmes techniques importants, ce qui permet de réaliser rapidement des essais. Cette méthode permet également d'éviter le piège des minima locaux .

La grande facilité à adapter l'algorithme en cas de modification des contraintes présente un certain atout. Il suffit en effet d'adapter la fonction de coût, le reste de l'algorithme restant le même.

Pour ce qui est des inconvénients, on retiendra particulièrement la difficulté de trouver les bonnes valeurs pour les différents paramètres à définir. En effet, leur influence sur l'efficacité de l'algorithme est importante et on ne peut les définir que de manière empirique, ce qui nécessite une certaine expérience dans le domaine.

La non-reproductibilité des résultats est un désavantage notable. En effet, la recherche étant totalement aléatoire, il est impossible d'en retracer le déroulement, même en relançant l'algorithme.

Enfin, l'algorithme ne s'arrête pas toujours sur la solution optimale, il peut s'arrêter sur une des configurations proches. Il est donc parfois nécessaire d'explorer les voisins immédiats de la solution finale.

Il est à noter également que si des études théoriques des principes sous-jacents à cet algorithme (les chaînes de Markov) permettent de prouver la convergence de l'algorithme, en pratique, on est obligé de borner le nombre d'itérations car le temps de convergence peut être quasi infini.

II.2.2.4 Adaptation de la méthode à notre problème.

Pour adapter cette méthode à notre problème, commençons par définir les caractéristiques d'une configuration quelconque. Pour un carré d'ordre n, une configuration $c=(x_1, x_2, \ldots, x_{n^2})$ doit avoir $x_1, \ldots, x_{n^2} \in [1, n^2]$ et $x_1 \neq x_2 \neq \ldots \neq x_{n^2}$. De plus, les deux diagonales principales doivent être conformes aux demandes de l'utilisateur, c'est-à-dire équivalentes à celles de départ aux permutations internes près et avec le noyau exactement équivalent à celui de départ.

La transition d'une configuration (c) vers une configuration voisine (c') se fait par permutation aléatoire, ce qui permet de garantir le maintien de ses caractéristiques sans devoir effectuer de contrôle sur l'unicité et sur la valeur des éléments. Lors de ces transitions, il faut toutefois empêcher la modification des cases centrales et vérifier qu'une éventuelle permutation sur les diagonales se fait conformément aux règles propres au problème qui nous occupe.

Enfin, il nous faut définir une fonction de coût adaptée. Pour rappel, la fonction de coût doit refléter la qualité de la configuration courante par rapport à une solution optimale. Dans notre cas, un carré magique d'ordre n doit respecter 2* N contraintes. A savoir, N sur les lignes et N sur les colonnes. De plus, dans chaque configuration, le coût sur une contrainte peut être facilement exprimé par la différence entre la somme linéaire de chacun des éléments de la ligne ou colonne et la somme magique d'ordre n. Une bonne fonction de coût global est la somme des valeurs absolues de ces différences. En effet, cette fonction a une valeur de 0, sa valeur minimum, dans le seul cas où le carré est magique et sa valeur

augmente lorsque les coûts sur les contraintes augmentent. Sa valeur nous donne donc bien une bonne représentation de la qualité de la configuration.

Pour illustrer, prenons un exemple de l'ordre 4.
Soit le carré suivant :

				D2 : 34 – 34 = 0
15	14	07	04	L1 : 40 – 34 = 6
10	12	13	01	L2 : 36 – 34 = 2
16	08	05	06	L 3 : 35 – 34 = 1
09	03	11	02	L 4 : 25 – 34 = -9
C1 :	C2 :	C3 :	C4 :	D1 : 34 – 34 = 0
50	37	36	13	
-	-	-	-	
34	34	34	34	
=	=	=	=	
16	3	2	-21	

Figure II-2 : La fonction de coût pour le recuit simulé

La valeur de la fonction de coût pour cette configuration est donc |6| + |2| + |1| |-9| + |-21| + |2| + |3| + |16| = 60

II.2.3 La recherche Tabou

Dans la méthode du recuit simulé, lorsque l'on effectue une transition entre deux configurations, les choix, tant de l'élément à modifier que de sa nouvelle valeur, sont effectués au hasard. La recherche se fait donc de manière totalement aléatoire. L'idée sous-jacente à la recherche Tabou est de tenter de diriger cette recherche pour accélérer la convergence en lui donnant un zeste d'intelligence.

Les informations mentionnées dans cette section sont issues de [Leclercq, 2005] et [Dréo et al, 2005b].

II.2.3.1 Principe de fonctionnement

Dans le principe, il s'agit d'explorer, pour une configuration donnée, son voisinage ou une partie de son voisinage et calculer la fonction de coût pour chaque configuration explorée. On passe alors de la configuration courante vers celle ayant la fonction de coût la plus optimisée, et ce quelle que soit sa valeur.

L'avantage de parfois se diriger localement vers des configurations de coût supérieur est de permettre de s'éloigner des optima locaux et de ne pas y rester piégé, l'inconvénient étant que l'on risque de boucler.

En effet, admettons que l'on passe d'une configuration c vers une configuration voisine c', laquelle est la meilleure configuration voisine mais possède néanmoins un coût global supérieur. Il est fort probable que lors de l'exploration du voisinage de c', on se retrouve à considérer le cas de c. Or, le coût de la configuration c étant inférieur, la possibilité que ce soit la configuration retenue n'est pas négligeable. L'algorithme entrerait ainsi dans une boucle infinie c → c' → c → c'→ …

Pour pallier à ce problème, un mécanisme de mémoire à court terme est utilisé : la liste Tabou, c'est d'ailleurs ce qui a donné son nom à la méthode. Ce mécanisme consiste à retenir les dernières configurations visitées et à empêcher d'y retourner. On va donc, à chaque itération, garder dans une liste les n dernières configurations visitées et les exclure de l'exploration des voisinages. Tout le problème étant de définir le nombre d'éléments que l'on met dans la liste. En effet, si le nombre d'éléments de la liste est inférieur au nombre d'éléments nécessaires pour se dégager de l'optimum local, l'algorithme va boucler.

En pratique, plutôt que de retenir des configurations complètes, une ou plusieurs caractéristiques des dernières solutions explorées seront retenues. Cela comporte plusieurs avantages, le premier, assez évident, étant de réduire la taille des données à retenir. Le second avantage est que cela permet d'exclure plusieurs configurations à la fois, ce qui limite d'autant le voisinage à explorer. Cette technique permet donc d'accélérer l'algorithme.

L'exclusion de nombreuses configurations via la technique ci-dessus étant fort contraignante, il est d'important d'ajouter également un mécanisme permettant de rendre un peu de souplesse à l'exploration du voisinage. Il y a donc lieu de mettre en place une seconde technique complémentaire : l'**aspiration**. Elle permet de passer outre à l'exclusion Tabou si le coût global est suffisamment amélioré. La notion « d'amélioration suffisante du coût » doit être définie de manière précise via une fonction propre à chaque problème spécifique.

Les paragraphes suivants vont permettre de détailler les différents points de cette méthode.

II.2.3.2 Les configurations et leur voisinage

Comme vu dans l'introduction de cette méthode, l'algorithme va progresser en explorant le domaine des solutions en passant d'une configuration à une autre qui lui est voisine.

Commençons donc par définir ce qu'est une configuration. Pour notre problème et pour un carré d'ordre n, on peut considérer l'ensemble des configurations (S) comme l'ensemble des carrés contenant les 1 à n^2 premiers nombres naturels, tous repris une et une seule fois, dont les deux diagonales principales sont celles demandées par l'utilisateur aux permutations internes près et dont le noyau est celui demandé par l'utilisateur.

Considérons l'ensemble des configurations voisines d'une configuration $s \in S$ comme un ensemble V tel que $V(s) \subset S$. Dans le cas qui nous occupe, une configuration voisine peut être définie comme une configuration s' pouvant être obtenue à partir de s en permutant deux de ses composants. L'exploration complète de V(s) pour notre problème consiste donc à permuter tous les éléments entre eux et à retenir la meilleure solution visitée.

En général, l'ensemble des éléments de V(s) ne sera pas considéré. En effet, que ce soit pour accélérer l'algorithme ou pour tenter de diriger la recherche, seul un sous-ensemble de V(s) sera exploré. En n'explorant pas tout le voisinage, il est possible que le meilleur mouvement possible ne soit pas choisi. Mais cet inconvénient est contrebalancé par le fait que cela permet d'explorer des mouvements qui n'auraient pas été retenus si l'ensemble de V(s) avait été exploré, ce qui va améliorer la diversification de la recherche.

Pour choisir ce sous-ensemble à explorer, plusieurs méthodes sont possibles. Une d'elles, proposée par Glover et que l'on peut retrouver plus en détail dans [Glover, 1997], consiste à classer chaque mouvement par ordre de qualité et à ne prendre en compte que les meilleurs pendant quelques itérations. Glover partait du principe qu'un bon mouvement devait le rester pour des solutions voisines. Il convient bien entendu de mettre à jour cette liste après quelques itérations, une fois que la solution s'est plus éloignée de la solution lors de laquelle la liste de mouvements a été créée. Un autre moyen de déterminer le sous-ensemble de V(s) à explorer est tout simplement de le générer aléatoirement.

II.2.3.3 La gestion de la liste Tabou

Cette liste est primordiale pour le fonctionnement de la méthode puisque sans elle, la recherche bouclerait lorsqu'elle devrait s'extraire d'un minimum local. Cependant, la gestion d'une telle liste n'est pas triviale.

En effet, la solution la plus intuitive est de sauvegarder les dernières configurations visitées. Cependant, cette solution est très inefficace, tant par le temps nécessaire à la vérification de l'ensemble de la solution à chaque itération, que par la place mémoire prise par une telle liste. De plus, l'interdiction pure et simple de repasser deux fois par la même solution peut s'avérer trop stricte dans certains cas.

Une solution généralement plus efficace, consiste à empêcher d'effectuer certains mouvements en stockant dans la liste des mouvements interdits ou des attributs de mouvements interdits. L'ordre de grandeur de l'ensemble des mouvements étant

généralement plus petit que celui des solutions, la taille de la mémoire utilisée est alors réduite et la rapidité de l'algorithme augmentée. De plus, même si une solution est visitée plusieurs fois, il est vraisemblable que la liste des mouvements interdits permettra de partir par un autre chemin. Le bouclage n'est néanmoins pas théoriquement impossible, même s'il reste très improbable.

Un autre élément important est le temps d'interdiction pour les mouvements Tabou. En effet, plus il est réduit et moins il y aura de mouvements interdits. On explorera donc les chemins empruntés plus en profondeur, ce qui permettra de trouver plus facilement un optimum. Par contre, il sera plus difficile de se dégager d'un minimum local dans lequel la recherche risque de boucler. A l'inverse, plus le temps d'interdiction est long et plus le nombre de mouvements interdits sera important. Cela permettra de se dégager de minima locaux relativement facilement en empruntant des chemins qui seraient normalement interdits. Par contre, on risque de rater des optimums en n'explorant pas assez les chemins choisis. La recherche risque dans ce cas d'être plus guidée par les mouvements autorisés que par la minimisation de la fonction de coût.

Une alternative consiste à modifier la durée de l'interdiction au cours de la recherche. La variation pouvant être aléatoire ou bien calculée en fonction de certains critères récoltés et calculés au cours de la recherche.

II.2.3.4 Les paramètres de direction de la recherche sur le long terme

Certaines techniques peuvent être ajoutées à la méthode de base pour diriger la recherche sur le long terme en permettant des phases d'intensification, et d'autres de diversification. L'intensification permet de concentrer la recherche autour de certaines valeurs prometteuses tandis que la diversification permet au contraire de relancer la recherche avec des mouvements peu utilisés. Ces techniques se basent notamment sur l'implémentation d'une mémoire à long terme.

Parmi les nombreuses possibilités connues, je citerai à titre d'exemple la technique se basant sur les fréquences dont le détail peut-être trouvé en [Dréo et al, 2005b]. Le principe est le suivant : si un mouvement donné se produit fréquemment, c'est vraisemblablement dû au fait que l'algorithme tourne sur un nombre restreint de solutions et qu'il y a donc lieu de diversifier la recherche. La technique consiste donc à affecter les mouvements d'un coefficient handicapant proportionnel à leur fréquence d'utilisation.

Une autre technique possible est de forcer un mouvement qui n'a pas été utilisé depuis longtemps, et ce, quel que soit le résultat en terme de minimisation de la fonction de coût. Cela permet de sortir d'un minimum local en favorisant des solutions qui ne le seraient jamais autrement.

II.2.3.5 Les avantages et inconvénients de la recherche Tabou

En ce qui concerne les performances, l'expérience montre que la recherche Tabou offre des résultats légèrement meilleurs que le Recuit Simulé. Cependant aucune base théorique ne peut étayer ces résultats.

Contrairement au recuit simulé, la convergence de la recherche Tabou ne peut pas être prouvée puisque l'on peut s'éloigner par moments de l'optimum local. Rien ne prouve non plus que la configuration courante à la fin de l'algorithme est la meilleure rencontrée. Il est donc nécessaire de la retenir.

Il est aussi à noter que les itérations de la recherche Tabou sont assez longues puisqu'il faut chaque fois explorer tout le voisinage immédiat ou tout au moins une partie importante de ce voisinage. Le nombre d'itérations est donc en général moins grand que dans la méthode du recuit simulé.

L'implémentation de la recherche Tabou est un peu plus complexe puisqu'il faut notamment ajouter la gestion de la liste Tabou, la mise en place de l'exploration systématique du voisinage immédiat et la fonction d'aspiration.

Enfin, il est à noter que les paramètres à choisir, comme la longueur de la liste Tabou, et les paramètres de la fonction d'aspiration sont cruciaux pour le bon fonctionnement de l'algorithme mais restent très dépendants du problème et donc déterminables uniquement par empirisme. Cela peut impliquer de nombreux essais avant de trouver les bonnes valeurs.

II.2.3.6 La recherche Tabou et les carrés magiques

Pour l'adaptation de la méthode à notre problème, la fonction de coût peut être la même que celle utilisée avec la méthode du Recuit Simulé, c'est-à-dire, la somme des valeurs absolues des différences entre la somme linéaire de tous les éléments de chaque ligne, colonne et diagonale principale et la somme magique d'ordre n.

Pour ce qui est de la notion de configuration, on peut également utiliser la même que pour le Recuit Simulé. Donc, pour un carré d'ordre n, une configuration $c=(x_1, x_2, \ldots, x_{n^2})$ doit avoir $x_1, \ldots, x_{n^2} \in [1, n^2]$ et $x_1 \neq x_2 \neq \ldots \neq x_{n^2}$. De plus, les deux diagonales principales devront être conformes aux demandes de l'utilisateur. On passera d'une configuration à l'autre par permutations, de manière à ne pas devoir contrôler l'unicité des éléments.

Lors de chaque itération, le voisinage immédiat doit être exploré. Par cette notion, sont considérées dans notre cas toutes les configurations obtenues en permutant chaque élément non fixé par tous les autres éléments non fixés. De même, les éléments des diagonales non fixés ne seront permutés qu'entre eux.

Pour limiter le temps d'exécution de chaque itération, la méthode proposée par Glover, mentionnée dans un sous-chapitre précédent, peut être utilisée.

En ce qui concerne la gestion de la liste Tabou, deux choses sont à définir : la forme et la taille. En ce qui concerne la forme, retenir les configurations complètes serait inefficace et coûteux en mémoire. Une forme possible serait de retenir l'emplacement et la valeur des deux éléments permutés, ce qui empêcherait de revenir à la configuration que l'on vient de quitter mais qui permettrait également de limiter le nombre de mouvements possibles en interdisant les autres configurations dont les valeurs retenues pour la permutation seraient à la même place.

En ce qui concerne la taille de la liste Tabou, elle ne peut être définie que par expérimentation en voyant si l'algorithme reste piégé dans des minima locaux.

II.2.4 La recherche adaptative

Cette méthode, proposée par Philippe Codognet et Daniel Diaz [Codognet, 2001], est issue de la famille des méthodes avec recherche locale, tout comme la recherche Tabou vue précédemment. Sa particularité est de diriger la recherche en calculant un coût sur les différentes variables plutôt que sur la configuration.

En effet, dans le cas de la recherche Tabou, le coût d'une configuration pourrait être calculé, par exemple, comme étant l'erreur globale sur toutes les contraintes. Avec la recherche adaptative, l'analyse sera affinée et le coût sera déterminé pour chacune des variables. Il est aussi possible de commencer par calculer les coûts sur les variables et en déduire le coût global de la configuration. Cette dernière approche sera d'ailleurs la plus fréquente car il n'est pas rare qu'il faille de toutes façons calculer le coût sur les variables pour pouvoir déterminer le coût global.

L'analyse du coût de chaque variable va permettre, à chaque itération, de déterminer la variable la plus mauvaise, celle qui viole le plus les contraintes. En modifiant la valeur de cette variable, on peut supposer que le coût global pourra être amélioré. Cela va permettre de limiter le voisinage à explorer par rapport à la recherche Tabou puisque la variable à modifier sera déjà connue. Dans un deuxième temps, il sera nécessaire de regarder quelle valeur du domaine peut être affectée à cette variable pour minimiser le coût global.

II.2.4.1 Projection de la fonction de coût sur les variables.

Les fonctions de coût sur les contraintes nous sont déjà connues, elles sont utilisées pour calculer le coût global sur les configurations dans les deux méthodes présentées précédemment. (La Recherche Tabou et le Recuit Simulé).

Pour projeter ces coûts sur les variables, la technique la plus largement utilisée est de sommer le coût des contraintes dans lesquelles la variable intervient. Cette méthode de projection des coûts suffit amplement pour un problème tel que celui des carrés magiques, mais la projection peut représenter un gros travail lors de la modélisation d'autres problèmes.

II.2.4.2 La gestion des minima locaux

Comme les autres techniques méta-heuristiques, la recherche adaptative intègre une technique permettant d'éviter le piège des minima locaux. Elle se base, comme la recherche Tabou, sur l'utilisation d'une liste. Celle-ci fonctionne de manière néanmoins fort différente de celle de la recherche Tabou.

Son fonctionnement est le suivant : chaque fois qu'une variable atteint un minimum local, elle est écartée pour quelques itérations de la recherche. On sait qu'une variable a une valeur correspondant à un minimum local si, dans une configuration donnée, aucune valeur du domaine qui lui est affectée ne permet de diminuer le coût global de la configuration.

Cette liste à elle seule ne permet pas d'éviter de tomber dans le piège des minima locaux, mais elle permet de détecter que la recherche est bloquée dans un minimum local. En effet, lorsque tous les éléments du carré se trouvent dans cette liste, cela signifie que le coût global de la configuration ne peut plus être diminué par permutations. On a donc atteint un minimum local.

Pour s'en extraire, une autre technique peut être utilisée : la remise à zéro partielle des variables. Cela donne une nouvelle configuration à partir de laquelle la recherche peut continuer.

II.2.4.3 Les mécanismes d'orientation de la recherche à long terme

Comme pour la recherche Tabou, la recherche adaptative va intégrer des techniques permettant de diriger la recherche sur le long terme en délimitant des phases d'intensification et des phases de diversification.

L'intensification est gérée grâce à l'utilisation de la liste des variables exclues de la recherche. En effet, en excluant une variable de la recherche, cette dernière va se concentrer autour de cette variable puisque sa valeur ne pourra plus être modifiée.

Pour ce qui est de la diversification, elle sera assurée par le mécanisme de remise à zéro. Pour permettre de mieux paramétrer cette diversification, la remise à zéro ne sera pas totale. En effet, elle se fera sur une partie des variables choisies aléatoirement. De même, la remise à zéro ne se fera pas quand toutes les variables seront marquées « Tabou», mais uniquement lorsqu'un seuil sera dépassé, c'est-à-dire lorsque le nombre de variables « Tabou » atteindra une valeur choisie.

Tous ces paramètres vont permettre de dimensionner les phases de diversification et d'intensification en fonction du problème, le choix de la valeur de ces paramètres devant se faire sur base d'expérimentation.

II.2.4.4 L'algorithme général

L'algorithme général de cette méthode est présenté ci-dessous. Il est donné indépendamment du problème posé.

Génération d'une configuration aléatoire
Faire
Calcul du coût de chaque variable dans la configuration.

Choisir aléatoirement V_{pire} parmi les variables ayant un coût maximum et n'étant pas marquées « Tabou ».

Remplacer V_{pire} par toutes les autres valeurs de son domaine et, pour chacune d'entre elles, calculer le coût global de la configuration.

Si une valeur du domaine améliorant le coût global est trouvée alors
Affecter aléatoirement à $V_{meilleure}$, une valeur choisie aléatoirement parmi celles minimisant le plus le coût global de la configuration.
Sinon
V_{pire} est marqué « Tabou » pour les prochaines itérations.

Si le nombre de variables marquées « Tabou » est supérieur au seuil alors
Remise à zéro partielle de la configuration courante
Tant que le coût global n'est pas annulé ou que le nombre maximum d'itérations n'est pas atteint.

II.2.4.5 Avantages et inconvénients de la recherche adaptative

Comme pour les autres méthodes méta-heuristiques, la convergence ne peut pas être prouvée. En effet, même si on converge à chaque itération vers un minimum local puisqu'une nouvelle valeur n'est affectée qu'en cas d'amélioration du coût global, rien ne garantit qu'un minimum global sera atteint pour la fonction de coût. De plus, pour garantir la fin de l'algorithme, le nombre d'itérations que l'on peut effectuer doit être borné, cela implique notamment que rien ne prouve que la recherche s'arrêtera sur la configuration de coût minimum explorée, il y a donc lieu de la retenir.

Par rapport à la recherche avec des tabous, cette méthode permet de bien cibler le voisinage à explorer grâce au choix de la variable à modifier, ce qui permet de gagner du temps lors des itérations.

Une des principales difficultés de cette méthode reste de trouver des fonctions de coût sur les contraintes qui soient faciles à projeter sur les variables.

Une autre difficulté consiste à bien dimensionner les paramètres dirigeant la recherche (nombre d'itérations d'exclusions, nombre de variables ré-initialisées lors des remises à zéro partielles et seuil des remises à zéro partielles). En effet, si ces paramètres sont mal choisis, ils peuvent fortement handicaper le déroulement de la recherche.

L'implémentation de cette méthode est moins complexe que celle de la recherche Tabou, notamment parce que les mécanismes de diversification de la recherche sont intégrés à l'algorithme global. Par contre, elle reste plus complexe à implémenter que le Recuit Simulé.

Un des principaux inconvénients de cette méthode, d'ailleurs commun à toutes les méthodes se basant sur des méta-heuristiques, reste sa non-reproductibilité. En effet, le même chemin ne sera jamais emprunté deux fois de suite, même avec des paramètres de recherche identiques.

Notons enfin qu'il n'est pas nécessaire, dans le problème qui nous occupe, de retenir la meilleure configuration visitée puisque le seul cas qui nous intéresse est celui où la fonction de coût prend une valeur nulle et que la recherche s'arrête dès qu'elle prend une telle valeur.

II.2.4.6 La recherche adaptative et les carrés magiques

La modélisation des carrés magiques pour cette méthode se fait assez naturellement.

La notion de configuration peut de nouveau être définie de la même manière que pour les méthodes précédemment étudiées. Soit une grille de n^2 éléments, chacun de ses éléments ayant une valeur comprise entre 1 et n^2 et toutes les valeurs comprises entre 1 et n^2 doivent être affectées une et une seule fois.

Le mouvement depuis une configuration vers une autre ne se fait que par permutation de deux valeurs, ce qui permet de ne pas devoir s'occuper de contrôler la valeur et l'unicité des éléments.

Pour ce qui est de la fonction de coût global, on peut utiliser également la même que pour les autres méthodes étudiées, à savoir la somme des valeurs absolues des coûts sur les contraintes.

Le calcul du coût sur chacune des variables se fait en additionnant le coût de la contrainte associée à sa ligne avec le coût de la contrainte associée à sa colonne. De nouveau, il n'est pas nécessaire de s'occuper des coûts des contraintes associées aux diagonales puisqu'elles sont supposées magiques.

II.2.5 La méthode choisie

Il est donc possible de modéliser le problème des carrés magiques avec les trois méthodes étudiées de façons assez similaires. En effet, les configurations visitées, les mouvements effectués et la fonction de coût global à minimiser sont les mêmes. Les trois méthodes doivent donc permettre de résoudre le problème.

Remarquons également que le problème est fortement connexe, ce qui signifie que n'importe quelle configuration peut être atteinte depuis une configuration de départ quelconque. Il existe en effet toujours une suite de permutations permettant de passer d'une configuration c quelconque à une configuration c'. Cela signifie que, quelle que soit la configuration de départ, si une solution existe, il est toujours possible de l'atteindre par une suite de permutations. Cette caractéristique est indispensable au fonctionnement des trois méthodes étudiées.

Il aurait été évidemment intéressant de toutes les implémenter pour pouvoir en comparer les performances. Néanmoins, comme on l'a vu lors de l'étude théorique, chaque méthode demande de régler des paramètres ou des mécanismes propres qui ont tout leur importance pour le bon fonctionnement de la recherche et qui ne peuvent être dimensionnés que par de nombreux jeux de tests.

Une seule a donc été implémentée : la recherche adaptative. Plusieurs raisons ont poussé à ce choix.

Tout d'abord, la projection sur les variables du coût sur les contraintes est particulièrement bien adaptée au problème des carrés magiques. En effet, le calcul du coût sur les contraintes, nécessaire au calcul du coût sur les variables, l'est aussi pour le calcul du coût global. Ce qui minimise la difficulté d'implémentation et la longueur de calcul du coût sur les variables.

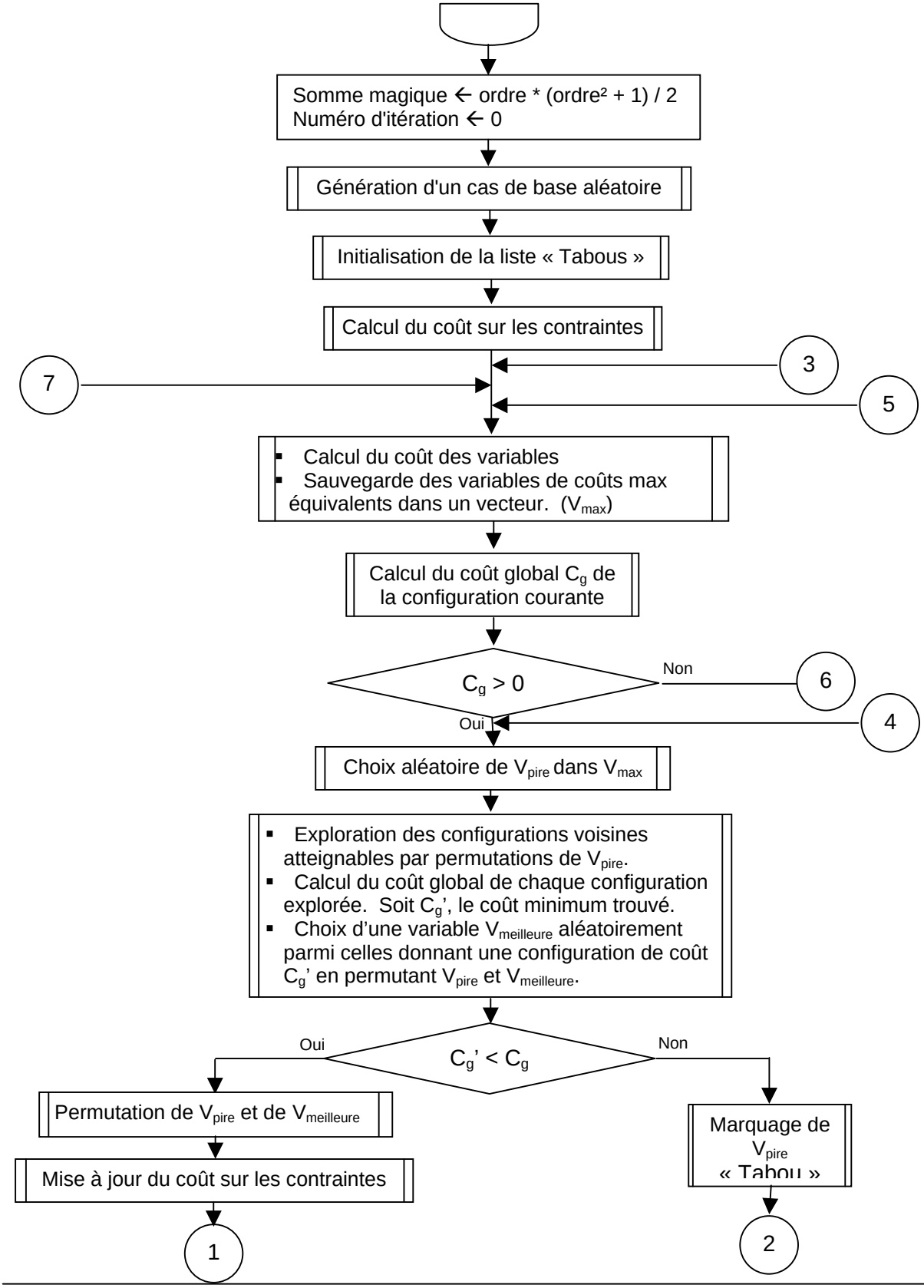
De plus, l'heuristique consistant à choisir la variable de coût maximum pour la permuter et essayer d'ainsi minimiser le plus rapidement possible le coût global fonctionne bien dans le cas des carrés magiques. En effet, la variable de coût maximum est très souvent celle dont la permutation permettra d'améliorer le plus le coût global. L'utilisation de cette heuristique va donc nous permettre de gagner facilement beaucoup de temps lors de l'exploration du voisinage des solutions, notamment par rapport à la recherche avec tabous. Cette heuristique a d'ailleurs été testée dans le cadre de ce travail. Cela a été fait en comparant les variables qui auraient été permutées pour minimiser le coût global par exploration complète du voisinage. Dans 80% des cas, la variable de coût maximum en faisait partie.

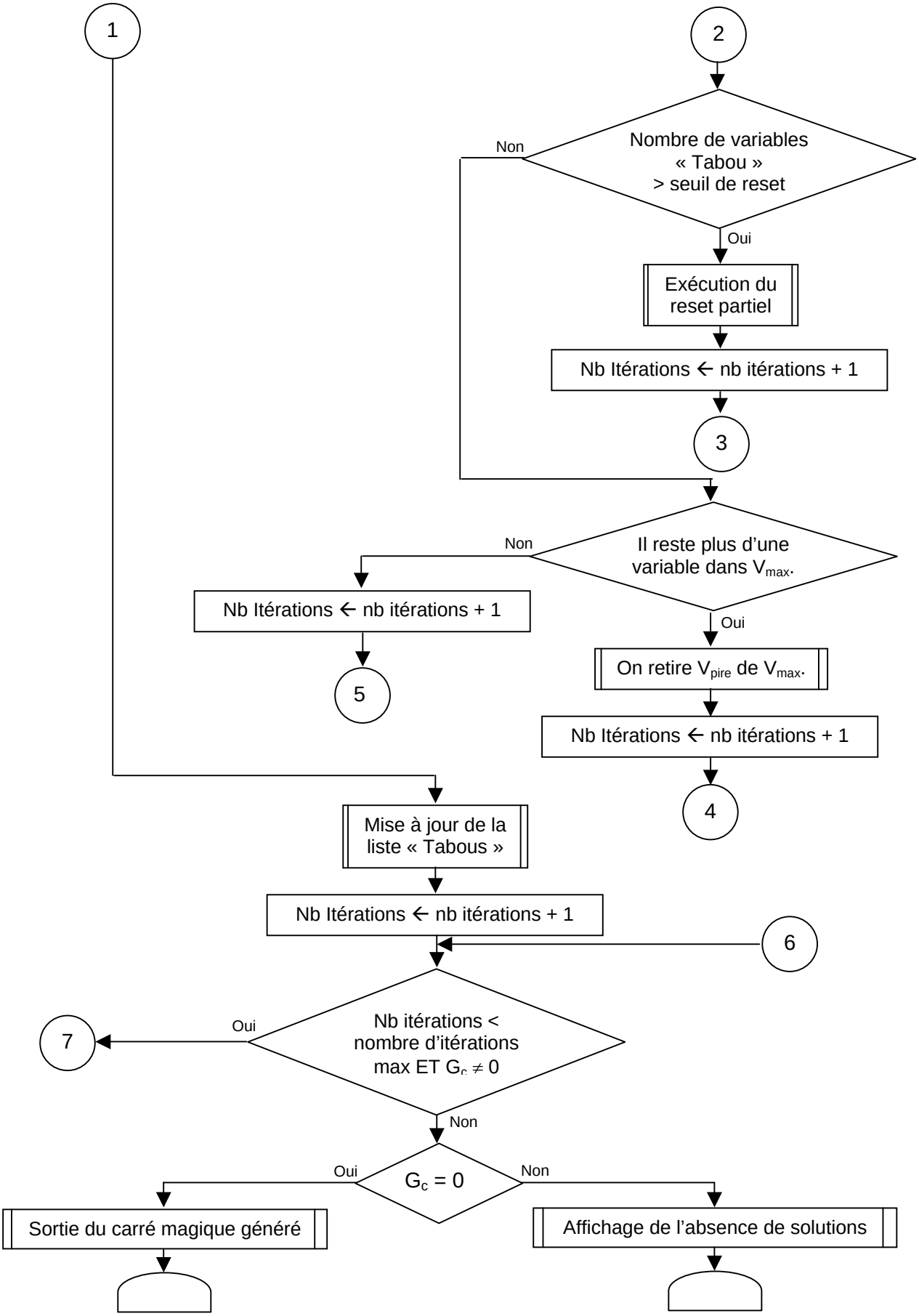
Et enfin, les résultats présentés par les auteurs de la méthode tendent à prouver qu'elle donne d'excellents résultats pour le type de problème qu'il s'agit de résoudre ici.

Chapitre III
L'IMPLEMENTATION DE LA RECHERCHE ADAPTATIVE

III.1 L'algorithme général et son fonctionnement

Voici l'algorithme du programme implémenté. Il permet la génération de carrés magiques de n'importe quel ordre à partir des diagonales fixées par hypothèse. Il utilise la recherche adaptative comme méthode de recherche.




1
2
Nombre de variables « Tabou » > seuil de reset
Non
Oui
Exécution du reset partiel
Nb Itérations ← nb itérations + 1
3
Il reste plus d'une variable dans V_{max}.
Non
Oui
Nb Itérations ← nb itérations + 1
5
On retire V_{pire} de V_{max}.
Nb Itérations ← nb itérations + 1
4
Mise à jour de la liste « Tabous »
Nb Itérations ← nb itérations + 1
6
Nb itérations < nombre d'itérations max ET $G_c \neq 0$
Oui
7
Non
$G_c = 0$
Oui
Non
Sortie du carré magique généré
Affichage de l'absence de solutions

III.2 Principe de fonctionnement et algorithme des différentes composantes du programme

Le plan de cette section a été défini de manière à correspondre à l'algorithme de la fonction générale présentée précédemment. Chaque composante représentée dans l'algorithme et qui mérite une présentation plus détaillée se retrouve ci-dessous.

Notons à ce stade que l'algorithme prend en compte l'hypothèse que les diagonales fournies par l'utilisateur sont correctes, c'est-à-dire qu'elles sont magiques et compatibles entre elles. Dans la pratique, le programme comprend également une fonction permettant de valider les hypothèses avant de lancer la recherche.

III.2.1 Quelques bases utilisées dans le reste de l'algorithme

Quelques parties d'algorithme sont ici présentées. Ce sont celles qui permettent de résoudre des points techniques particuliers qui apparaissent fréquemment dans les autres parties d'algorithme présentées dans les points suivants.

Commençons par fixer la notion d'index : l'index d'un élément correspond à la case du carré dans laquelle il se trouve. Ces cases sont numérotées dans l'ordre naturel et la première case a un index 0. Pour l'ordre 4, cela nous donne donc un carré avec les index suivants :

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

Figure III-1 : Les index d'un carré d'ordre 4

III.2.1.1 Déterminer si un élément fait partie de la diagonale descendante.

Les éléments se trouvant dans la diagonale descendante sont les suivants :

0	1	2	3	4
5	6	7	8	9
10	11	12	13	14
15	16	17	18	19
20	21	22	23	24

Figure III-2 : La diagonale descendante

Chaque élément de cette diagonale est à n+1 case du précédent, et son premier élément a un index de 0. Les éléments de cette diagonale auront donc toujours un index qui sera un multiple de (n + 1). Pour vérifier si un élément d'index i fait partie de cette diagonale, il faut donc vérifier que le reste de la division entière de i par (n + 1) est égale à 0.

On notera également que pour déterminer l'index j du $x^{ème}$ élément de la diagonale, il suffit d'appliquer la formule suivante :

$j = (x - 1) * (n + 1)$

III.2.1.2 Déterminer si un élément fait partie de la diagonale montante.

Les éléments se trouvant dans la diagonale montante sont les suivants :

0	1	2	3	4
5	6	7	8	9
10	11	12	13	14
15	16	17	18	19
20	21	22	23	24

Figure III-3 : La diagonale montante en vert

Chaque élément de cette diagonale est à n-1 case du précédent, et son premier élément a un index de n - 1. Les éléments de cette diagonale auront donc toujours un index qui sera un multiple de (n - 1). Pour vérifier si un élément d'index i fait partie de cette diagonale, il faut donc vérifier que le reste de la division entière de i par (n - 1) est nulle, mais également que i est différent de 0 ou de $n^2 - 1$, ces deux valeurs étant les seuls multiples de n – 1 correspondant à une valeur possible de i ne se trouvant pas sur la diagonale montante.

Pour déterminer l'index j du $x^{ème}$ élément de la diagonale, il faut appliquer la formule suivante :

$j = x * (n - 1)$

III.2.1.3 Déterminer si un élément fait partie du noyau

Pour rappel, le noyau du carré est la case centrale pour un carré d'ordre impair et les 4 cases centrales pour un carré d'ordre pair. Les valeurs s'y trouvant ne peuvent jamais être modifiées durant la recherche

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

Figure III-4 : Le noyau d'un carré d'ordre pair

0	1	2	3	4
5	6	7	8	9
10	11	12	13	14
15	16	17	18	19
20	21	22	23	24

Figure III-5 : Le noyau d'un carré d'ordre impair

Pour les carrés d'ordre impair, la case centrale est celle dont l'index est la moitié de $n^2 - 1$ où n est l'ordre du carré. Pour les carrés d'ordre pair, il faut prendre les deux éléments centraux sur les deux diagonales. La procédure pour choisir le $X^{ème}$ élément d'une diagonale a déjà été exposée précédemment (Voir. III.2.1.1 et III.2.1.2). Voici l'algorithme correspondant :

```
Algorithme déterminant si l'élément d'index i se trouve dans le noyau d'un carré d'ordre n
{
    si n est pair alors
    {
        si i = (n / 2 – 1) * (n + 1) OU i = n / 2 * (n - 1) OU i = (n / 2 + 1) * (n - 1) OU i = n / 2 * (n + 1) alors
            i est dans le noyau
    }
    sinon
    {
        si i = (n² - 1) / 2 alors i est dans le noyau
    }
}
```

Figure III-6 : Algorithme déterminant l'appartenance de l'élément i au noyau du carré

III.2.2 Génération d'un cas de base aléatoire

Il s'agit de pouvoir générer une configuration aléatoire valide, ce sera le point de départ de l'algorithme. En d'autres termes, il s'agit de générer un carré contenant les 1 à n^2 premiers nombres entiers, tous repris une et une seule fois. De plus, les diagonales de ce carré doivent être celles imposées par hypothèse.

La méthode suivante est utilisée :

1. Génération du carré naturel d'ordre n. C'est-à-dire un carré d'ordre n dans lequel les nombres de 1 à n^2 sont mis dans leur ordre naturel.

01	02	03	04
05	06	07	08
09	10	11	12
13	14	15	16

Figure III-7 : Le carré naturel

2. Utilisation de l'algorithme du mélangeur de Fisher-Yates, lequel permet de permuter de manière aléatoire les éléments du carré.

```
Soit un carré c d'ordre n
{
    si (i >1)
    {
        pour i = n – 1 et tant que i > 0 par pas de –1sur i
        {
            j = rand() * (i + 1);
            permuter c[i] et c[j]
        }
    }
}
```

Figure III-8 : Mélangeur de Fisher-Yates

3. Les diagonales, imposées par hypothèse, sont reconstituées. Pour ce faire, on va parcourir les diagonales pour chaque élément, rechercher la valeur demandée par l'utilisateur dans le reste du carré. Une fois cette valeur trouvée, elle sera permutée avec celle se trouvant dans l'élément de la diagonale en cours d'analyse.

```
Recomposition des diagonales pour la génération du cas de base :
    Soit un carré c d'ordre n
    Soit diag1et diag2, les diagonales demandées
    {
        pour i = 1 et tant que i < n par pas de 1
        {
            valeurAChercher1 = diag1[i]
            valeurAChercher2 = diag2[i]

            pour j = 1 et tant que j < n² et que valeurAChercher1 non trouvée par pas de 1sur j
            {
                si  c[j] = valeurAChercher1    alors
                {
                    j = indexAPermuter1
                    valeurAChercher1 trouvée
                }
            }
            permuter c[ indexAPermuter1] et i^ème élément de la diagonale1 de c

            pour j = 1 et tant que j < n² et que valeurAChercher2 non trouvée par pas de 1sur j
            {
                si  c[j] = valeurAChercher2    alors
                {
                    j = indexAPermuter2
                    valeurAChercher2 trouvée
                }
            }
            permuter c[ indexAPermuter2] et i^ème élément de la diagonale1 de c
        }
    }
```

Figure III-9 : Recomposition des diagonales lors de la génération du cas de base

Pour un carré d'ordre n, le calcul de la complexité de l'algorithme permettant la génération d'un cas aléatoire est donc de n^2 pour la création du carré naturel plus n pour le mélangeur de Fisher-Yates plus $n * (n^2 + n^2)$ pour la reconstitution des diagonales, ce qui fait une complexité de $n^2 + n + 2n^3$. Ce qui nous donne une complexité théorique de l'ordre $O(n^3)$.

III.2.3 La gestion de la liste des variables marquées « Tabou »

Comme vu dans le chapitre II avec l'explication théorique de la méthode de la recherche adaptative, les variables peuvent être marquées « Tabou », c'est-à-dire exclues de l'algorithme pour un certain nombre d'itérations.

Nous allons également utiliser ce mécanisme pour fixer les éléments du noyau du carré. Pour ce faire, nous allons les marquer « Tabou » pour toute la durée de l'algorithme. Elles seront ainsi exclues à chaque itération, ce qui les fixera naturellement.

Pour représenter la liste des variables marquées « Tabou », un vecteur de taille n^2 sera utilisé, soit V_{tabou}. Dans ce vecteur, on aura V_{tabou} [i] = nombre d'itérations d'exclusion restantes pour l'élément d'index i du carré. Une valeur de V_{tabou} [i] égale à 0 indiquant que la variable d'indexe i n'est pas marquée « Tabou »

Pour éviter de devoir parcourir inutilement le vecteur, on utilisera également une variable permettant de connaître à tout moment le nombre d'éléments marqués « Tabou ».

III.2.3.1 L'initialisation de la liste « Tabou »

L'initialisation de la liste consiste donc à affecter 0 à toutes les valeurs de V_{tabou} sauf celles correspondant au noyau qu'il faut initialiser au nombre d'itérations maximum + 1.

```
Initialisation de V_Tabou
    Soit n, l'ordre du carré
{
    pour i = 0 et tant que i < n² par pas de 1 sur  i
    {
        V_Tabou [i] = 0
    }

    si n est pair alors,
    {
        V_Tabou [(n / 2 – 1) * (n + 1)] = nombre d'itérations maximum + 1
        V_Tabou [n / 2 * (n - 1)] = nombre d'itérations maximum + 1
        V_Tabou [(n / 2 + 1) * (n - 1)] = nombre d'itérations maximum + 1
        V_Tabou [n / 2 * (n + 1)] = nombre d'itérations maximum + 1
        NombreVariablesTabous = 4
    }
    sinon
    {
        V_Tabou [(n / 2 – 1) * (n + 1)] = nombre d'itérations maximum + 1
        NombreVariablesTabous = 1
    }
}
```

Figure III-10 : Algorithme permettant l'initialisation de la liste "Tabou"

La complexité de l'algorithme permettant l'initialisation de la liste « Tabou » est de $O(n^2)$

III.2.3.2 La mise à jour de la liste « Tabou »

La liste des variables marquées « Tabou » doit être mise à jour entre chaque itération. Cette mise à jour consiste à diminuer de 1 toutes les valeurs de V_{Tabou} supérieures à 0.

La complexité de cette partie de l'algorithme est de l'ordre $O(n^2)$

III.2.3.3 Marquage d'une variable « Tabou »

Pour marquer « Tabou » la variable i, une valeur positive sera attribuée à $V_{Tabou}[i]$. Cette valeur correspond au nombre d'itérations d'exclusion. Ce nombre sera discuté dans un chapitre ultérieur.

La complexité théorique de cette partie de l'algorithme est une constante, elle est indépendante de l'ordre de grandeur du carré.

III.2.3.4 Suppression du marquage « Tabou » d'une variable

Cela permet de réintégrer la variable à l'algorithme, et ce, quel que soit le nombre d'itérations d'exclusion qu'il lui reste. La première chose à faire est de vérifier si la variable est « Tabou ». Si elle l'est, la valeur 0 est attribuée à $V_{Tabou}[i]$ et on décrémente de 1 le nombre de variables « Tabou».

La complexité théorique de cette partie de l'algorithme est une constante, elle est indépendante de l'ordre de grandeur du carré.

III.2.4 Calcul du coût sur les contraintes

Les contraintes sont les équations à respecter pour que les sommes sur chaque ligne, colonne et diagonale principale soient magiques, il y a donc une contrainte par ligne, colonne et diagonale principale. Le coût sur une contrainte est la différence entre la somme de tous les éléments de la ligne, colonne ou diagonale principale et la somme magique. Ce qui nous donne $v_1 + v_2 + \ldots + v_n - s$. Avec s = somme magique, soit $\boldsymbol{n\,(n^2 + 1) / 2}$ où n est l'ordre du carré.
Voici un exemple de calcul du coût sur les contraintes pour un carré d'ordre 4 :

				0
07	05	06	08	-8
16	04	14	15	15
11	03	13	12	5
09	02	01	10	-12
9	-20	0	11	0

Figure III-11 : Le calcul du coût sur les contraintes

Pour un carré d'ordre n, on va donc devoir calculer 2 * n coûts sur les contraintes puisqu'il est inutile de s'occuper des contraintes sur les diagonales, lesquelles sont magiques par hypothèse. Elles seront toutes calculées via un parcours unique de tous les éléments du carré.

Le procédé utilisé est le suivant :

1. Initialisation des 2*n coûts à une valeur de –s.
2. Tous les éléments du carré seront parcourus, et, pour chaque élément, une mise à jour du coût des différentes contraintes dans lesquelles il intervient sera

effectuée. La mise à jour consiste à ajouter la valeur de l'élément au coût actuel de chaque contrainte dans lesquelles l'élément intervient.

Ce procédé nécessite de savoir, pour chaque élément parcouru, quelles sont les contraintes à mettre à jour. Pour cela, on va utiliser l'index dans le carré de l'élément en cours d'analyse. Comme vu précédemment, l'index est en base 0.

Il faut donc effectuer :

- Le calcul de la ligne sur laquelle se trouve l'élément

  Le numéro de la ligne sur une base 0 est égale au résultat de la division entière de l'index par l'ordre du carré. Ainsi, si dans le carré présenté en figure 11, on prend l'élément "04", son index est 5, son numéro de ligne est donc le résultat de la division entière de 5 par l'ordre du carré, soit 4. Cela donne 1, qui correspond à la deuxième ligne puisque l'on travaille sur une base de 0.
- Le calcul de la colonne sur laquelle se trouve l'élément

  Le numéro de la colonne sur une base 0 est égale au reste de la division entière (modulo) de son index par l'ordre du carré. Ainsi, si dans le carré présenté en figure 11, on reprend l'élément "04", son index étant 5, le reste de la division entière de cet index par l'ordre du carré, soit 4, est égal à 1, qui correspond bien à la seconde colonne puisque l'on travaille sur une base 0.
- Pour mettre à jour les contraintes associées à l'élément "04" du carré présenté en figure 11. La valeur 4 sera ajoutée au coût de la contrainte de la seconde ligne et de la seconde colonne.

Calcul du coût sur les contraintes.

```
Soit s, la somme magique d'ordre n
Soit c, la configuration actuelle, d'ordre n
{
    pour i = 0 et tant que i < n par pas de 1 sur i
    {
        coutSurLesLignes[i] = s * -1
        coutSurLesColonnes[i] = s * -1
    }

    pour i = 0 et tant que i < n² par pas de 1 sur i
    {
        ligneDeElement = Valeur entière de la division de i par n
        coutSurLesLignes[ligneDeElement] = coutSurLesLignes[ligneDeElement] + c[i]

        colonneDeElement = reste de la division entière de i par n
        coutSurLesColonnes[colonneDeElement] = coutSurLesColonnes[colonneDeElement] + c[i]
    }
}
```

Figure III-12 : Algorithme du calcul sur les contraintes

La complexité de cette partie de l'algorithme est de $n + n^2$, ce qui donne une complexité théorique de $O(n^2)$

III.2.5 Le calcul du coût sur les variables

Cette partie consiste à calculer le coût de chaque variable dans la configuration actuelle en fonction des coûts sur les contraintes. Ce coût sur chaque variable est la valeur absolue de la somme des coûts sur les contraintes dans lesquelles la variable intervient, c'est-à-dire les coûts sur les contraintes de sa ligne et de sa colonne.

Pour éviter de cumuler artificiellement des erreurs de signes opposés, il est obligatoire d'utiliser la valeur absolue de la somme et pas la somme des valeurs absolues. De plus, le passage à la valeur absolue permet de n'avoir que des coûts positifs.

Le principe de cette partie de l'algorithme est simple. Il faut parcourir les éléments du carré, et, pour chacun d'entre eux, faire la somme des coûts sur les contraintes pour lesquels il intervient. Pour savoir quelles sont les contraintes concernées, il faut calculer la ligne et la colonne où se trouve l'élément de la même manière que lors du calcul du coût sur les contraintes. Il est inutile de tenir compte des éventuels coûts sur les diagonales puisqu'elles sont magiques par hypothèse.

De plus, lors du calcul, il faudra retenir dans un vecteur V_{max}, les variables ayant un coût équivalent au coût maximum rencontré.

Cela nous donne l'exemple suivant :

07	05	06	08	-8
16	04	14	15	15
11	03	13	12	5
09	02	01	10	-12
9	-20	0	11	0

0

|-8+9|	|-8-20|	|-8-0|	|-8+11|
|9+15|	|15-20|	|15+0|	|15+11|
|5+9|	|5-20|	|5+0|	|5+11|
|9-12|	|-12-20|	|0-12|	|11-12|

=

1	28	8	3
24	5	15	26
14	15	5	16
3	32	12	1

Figure III-13 : Calcul du coût sur les variables

Calcul du coût sur les variables.

```
Soit s, la somme magique d'ordre n
Soit c, la configuration actuelle, d'ordre n
Soit coutSurLesLignes, le coût sur les contraintes de chaque ligne
Soit coutSurLesColonnes, le coût sur les contraintes de chaque colonne
{
    coutVariableMax = -1

    pour i = 0 et tant que i < n² par pas de 1 sur i
    {
        si i n'est pas « Tabou »
        {
            ligneDeElement = Valeur entière de la division de i par n
            coutVariable = coutSurLesLignes[ligneDeElement]

            colonneDeElement = reste de la division entière de i par n
            coutVariable  = coutVariable  + coutSurLesColonnes[colonneDeElement]
        }
        coutVariable = |coutVariable|

        Si coutVariable = coutVariableMax alors Ajouter i à Vmax

        Si coutVariable > coutVariableMax alors
        {
            coutVariableMax = coutVariable
            Vider Vmax
            Ajouter i à Vmax
        }
    }
}
```

Figure III-14 : Algorithme du calcul du coût sur les variables

La complexité théorique de cette partie de l'algorithme de $O(n^2)$ puisqu'il faut parcourir une fois tous les éléments du carré.

III.2.6 Calcul du coût global d'une configuration

Cette partie consiste à évaluer la qualité globale d'une configuration. Ce coût est simplement calculé en additionnant la valeur absolue des coûts sur les contraintes. On additionne des valeurs absolues pour éviter que des coûts sur les contraintes de signes opposés ne s'annulent. De plus, on additionne bien les coûts sur les contraintes, et non les coûts sur les variables, ce qui n'aurait pas de sens. On remarquera que le carré est magique si, et seulement si, le coût global de la configuration est égal à 0.

Etant donné la simplicité de cette partie, je ne développerai pas ici l'algorithme dans son détail.

Voici un exemple de calcul du coût global.

				0
07	05	06	08	-8
16	04	14	15	15
11	03	13	12	5
09	02	01	10	-12
9	-20	0	11	0

Coût global :
|-8| + |15| + |5| + |-12| + |11| + |0| + |-20| + |9| | = **80**

Figure III-15 : Le calcul du coût global de la configuration

La complexité de l'algorithme du calcul du coût global est de 2n, c'est-à-dire le nombre des contraintes à additionner. Sa complexité théorique est donc de l'ordre O(n).

III.2.7 Mise à jour du coût sur les contraintes

Le coût sur les contraintes est très fréquemment modifié lors de l'exécution de l'algorithme. Or, le calcul complet demande de parcourir tous les éléments du carré, ce qui peut s'avérer très long pour les ordres de grandeur les plus grands.

Les seuls mouvements intervenant entre les différents éléments du carré sont des permutations. Il est donc possible de savoir à chaque mouvement quelles sont les contraintes dont le coût a été modifié, à savoir celles correspondant à la ligne et à la colonne des deux éléments. Il est inutile de tenir compte des modifications possibles sur les coûts des contraintes suivant les diagonales puisqu'elles sont toujours magiques.

De plus, les coûts sur les contraintes qu'il est nécessaire de mettre à jour ne doivent pas être entièrement re-calculés. Il suffit de leur soustraire la valeur de l'élément avant permutation et d'ajouter la valeur de l'élément après permutation.

Cette méthode va nous permettre de mettre à jour les coûts sur les contraintes sans devoir parcourir les éléments du carré, ce qui nous permet de calculer les contraintes avec une complexité théorique en $O(1)$ au lieu de $O(n^2)$.

Voici l'algorithme correspondant :

Algorithme de mise à jour du coûts sur les contraintes :

Soit coutLignes, les coûts des contraintes sur les lignes.
Soit coutColonnes, les coûts des contraintes sur les colonnes..
Soit c, un carré d'ordre n
Soit indexVar1, l'index de la première variable.
Soit valVar1, la valeur de la première variable.
Soit indexVar2, l'index de la seconde variable.
Soit valVar2, la valeur de la seconde variable.
Rem : les index des variables sont ceux après permutation.

```
{
    indexCoutLigneVar1 = résultat de la division entière de indexVar1 par n
    indexCoutLigneVar2 = résultat de la division entière de indexVar2 par n

    indexCoutColonneVar1 = reste de la division entière de indexVar1 par n
    indexCoutColonneVar2 = reste de la division entière de indexVar2 par n

    coutLignes[indexCoutLigneVar1] = coutLignes[indexCoutLigneVar1] – valVar2 + valVar1
    coutLignes[indexCoutLigneVar2] = coutLignes[indexCoutLigneVar1] – valVar1 + valVar2

    coutColonnes[indexCoutLigneVar1] = coutColonnes [indexCoutLigneVar1] – valVar2 + valVar1
    coutColonnes [indexCoutLigneVar2] = coutColonnes [indexCoutLigneVar1] – valVar1 + valVar2
}
```

Figure III-16 : Algorithme de la mise à jour des coûts sur les contraintes

III.2.8 Exploration d'une partie du voisinage d'une configuration et calcul de la configuration voisine de coût global minimum

Cette partie de l'algorithme consiste à explorer une zone bien déterminée du voisinage d'une configuration c et à retenir une configuration voisine c' de coût global minimum.

Pour ce faire, un des éléments e_i du carré sera permuté avec toutes les autres variables compatibles. Soit c_{ej}, la configuration voisine de c générée par permutation de e_i et de e_j. Par variables compatibles, on entend que les variables marquées « Tabou » ne seront pas permutées. De plus, si la variable se trouve sur une des diagonales, elle ne pourra être permutée qu'avec les autres variables de cette même diagonale. De même, une variable n'appartenant pas à une diagonale ne pourra pas être permutée avec une variable se trouvant sur les diagonales.

Pour chaque configuration c_{ej} visitée, son coût global sera calculé suivant la méthode vue au point 2.5 (calcul du coût global d'une configuration). Soit Cg_{min} le coût global minimum calculé lors de l'exploration du voisinage.

Tous les éléments e_s tels que c_{es} ait un coût global égal à Cg_{min} seront retenus dans un vecteur $V_{moins\ cher}$.

La configuration c' recherchée sera obtenue en permutant e_i avec un des éléments de $V_{moins\ cher}$ choisi aléatoirement.

Recherche de la configuration voisine de coût minimum.

```
Soit un carré d'ordre n avec une configuration actuelle c
Soit indexVariableAPermuter, la variable que l'on va essayer de permuter
{
    coutGlobalMinimum = +infini
    pour  i = 0 et tant que i < n² et par pas de 1 sur i
    {
        Si i = indexVariableAPermuter alors passer directement à l'itération suivante.

        Si c[i] est marquée « tabou » alors passer directement à l'itération suivante.

        Si i se trouve sur la diagonale 1 et indexVariableAPermuter ne s'y trouve pas alors
            Passer directement à l'itération suivante

        Si i se trouve sur la diagonale 2 et indexVariableAPermuter ne s'y trouve pas alors
            Passer directement à l'itération suivante

        Permuter c[i] et c[indexVariableAPermuter]
        Mettre à jour ces contraintes après permutation.

        Calculer le coût global de la nouvelle configuration c'

        Si le coût global de c' = coutGlobalMinimum
            Ajouter i à Vmin

        Si le coût global de c' < coutGlobalMinimum
        {
            Vider Vmin
            Ajouter i à Vmin
            CoutGlobalMinimum = coût global de c'
        }

        Permuter c[indexVariableAPermuter] et c[i] pour retourner à la configuration c.
        Mettre à jour ces contraintes après permutation.
    }

    Choix de Vmeilleure aléatoirement dans Vmin
}
```

Figure III-17 : Recherche de la configuration voisine de coût minimum

La complexité de cette partie de l'algorithme est calculée de la manière suivante. La recherche est effectuée pour chacun des éléments du carré, soit n^2 fois. Lors de chaque itération, toutes les opérations la composant s'exécutent en un temps constant, sauf le calcul du coût global qui est de l'ordre O(n). Cela nous donne donc une complexité globale théorique de l'ordre de $O(n^3)$.

III.2.9 La remise à zéro partielle

Il s'agit ici de re-générer aléatoirement une partie du carré. C'est un mécanisme important dans le cadre de l'évitement du piège du minimum local. Cette partie est déclenchée lorsque le nombre de variables marquées « Tabou » du carré atteint un certain nombre, dont nous discuterons la valeur dans un chapitre suivant.

La remise à zéro partielle consiste donc à choisir aléatoirement certains éléments du carré et à les mélanger de manière tout aussi aléatoire. Le nombre d'éléments que l'on va mélanger est un autre paramètre important dont le dimensionnement sera abordé dans un chapitre suivant.

La remise à zéro partielle se fait donc en deux parties. Dans la première partie, les variables à mélanger sont choisies aléatoirement. Lors de ce choix, il faut faire attention à ne pas choisir d'éléments du noyau. Les index des variables retenues seront stockés.

Dans un deuxième temps, on procèdera à des permutations aléatoires. Il faut alors désigner la première variable à permuter en choisissant un index aléatoire parmi celles retenues au point précédent. Pour choisir la deuxième variable, c'est un peu plus complexe. En effet, cela va dépendre de la position de la première dans le carré. Si la première ne se trouve pas sur une des diagonales, on choisit aléatoirement une autre variable sélectionnée au point précédent en prenant soin de vérifier qu'elle n'est pas non plus sur une diagonale. Si la première se trouve sur une diagonale, il faut procéder autrement car rien ne prouve qu'une seconde variable se trouvant sur la même diagonale et ayant été pré-sélectionnée sera trouvée facilement. On va donc choisir aléatoirement une autre variable sur la diagonale plutôt que dans l'ensemble des variables pré-sélectionnées juste avant. On vérifiera cependant que la variable choisie ne fait pas partie du noyau.

Pour chaque variable qui aura été permutée, il faut vérifier si elle est marquée « Tabou » ou pas. Si c'est le cas, il faudra la supprimer de la liste. Il n'y a en effet plus aucune raison de l'exclure puisqu'elle n'intervient plus pour les mêmes contraintes. Par contre, les variables marquées « Tabou » et qui ne sont pas permutées resteront exclues. En effet, rien ne laisse supposer que la configuration soit suffisamment modifiée que pour que l'on puisse les réintégrer. Néanmoins, ce point peut être soumis à discussion, notamment si on augmente le nombre de variables à mélanger lors de la remise à zéro partielle.

Cela nous donne l'algorithme suivant :

```
Algorithme de reset partiel du carré

Soit un carré c d'ordre n
Soit nbVariables, le nombre de variables à permuter.

{
    pour i = 0 et tant que i < nbVariables et par pas de 1 sur i
    {
        Faire
            Génération aléatoire d'un index du carré.
        Tant que l'index généré correspond à un élément se trouvant dans le noyau

        ListeIndexReset[i] = index généré
     }

    Pour i = 0 et tant que i < nombre de permutations par pas de 1 sur i
    {
        Génération aléatoire de indexVar1, un nombre entre 0 et nbVariables – 1

        Si indexVar1 correspond à un index de la diagonale montante alors
        {
            Faire
                Générer aléatoirement indexVar2, un nombre entre 0 et n-1
            Tant que le indexVar2ème élément de la diagonale montante ne fait pas partie du noyau.

            Permuter c[listeIndexReset[indexVar1]] et c[(indexVar2 + 1) * (n-1)]
            Mettre à jour le coût sur les contraintes.
            Retirer c[listeIndexReset[indexVar1]] et c[(indexVar2 + 1) * (n-1)] de la liste « Tabou »
        }

        Si indexVar1 correspond à un index de la diagonale descendante alors
        {
            Faire
                Générer aléatoirement indexVar2, un nombre entre 0 et n-1
            Tant que le indexVar2ème élément de la diagonale descendante ne fait pas partie du noyau.

            Permuter c[listeIndexReset[indexVar1]] et c[indexVar2 * (n+1)]
            Mettre à jour le coût sur les contraintes.
            Retirer c[listeIndexReset[indexVar1]] et c[indexVar2 * (n+1)] de la liste « Tabou »
        }

        Si indexVar1 ne correspond pas à un index se trouvant sur une diagonale alors
        {
            Faire
                Génération aléatoire de indexVar2, un nombre entre 0 et nbVariables – 1
            Tant que listeIndexReset[indexVar2] ne correspond pas à un index se trouvant sur une diagonale

            Permuter c[listeIndexReset[indxeVar1]] et c[listeIndexReset[indxeVar2]]
            Mettre à jour le coût sur les contraintes.
            Retirer c[listeIndexReset[indxeVar1]] et c[listeIndexReset[indxeVar2]] de la liste « Tabou »
        }
    }
}
```

Figure III-18 : Algorithme de remise à zéro partielle de la configuration courante

La complexité de cette partie de l'algorithme est proportionnelle au nombre de permutations que l'on va effectuer. Dans la pratique, l'algorithme effectuera $nbVariables^2$ permutations. Sachant que $nbVariables \leq n$, la complexité théorique est donc $O(n^2)$.

III.2.10 La complexité globale de l'algorithme.

Le calcul de la complexité globale de l'algorithme se fait en considérant les complexités calculées pour les différentes parties de l'algorithme. Le constat est que la partie de l'algorithme qui ne s'exécute qu'une fois (génération du cas de base, initialisation de la liste « Tabou » et calcul du coût sur les contraintes) est d'une complexité théorique de l'ordre de $O(n^3)$, tandis que la partie itérative est d'une complexité théorique de l'ordre de $O(n^3)$.

Pour la partie ne s'exécutant qu'une fois, c'est la génération du cas de base qui est la plus lente. Je n'ai trouvé aucune méthode permettant d'en réduire la complexité théorique.

Quant à la partie itérative, sa complexité est déterminée par l'exploration du voisinage, laquelle est la seule partie de l'algorithme ayant une complexité supérieure à $O(n^2)$. Néanmoins, dans la pratique, on constate que l'ordre de grandeur réel est légèrement inférieur puisqu'il dépend également du nombre de variables marquées « Tabou ».

III.2.11 Le code source

Un code source en C implémentant l'algorithme présenté dans ce chapitre se trouve en annexe (.A.1) de ce travail. C'est sur ce code source que les différents jeux de test ont été effectués.

Chapitre IV
LES DIFFÉRENTS PARAMÈTRES ET LEUR DIMENSIONNEMENT

Comme nous l'avons vu lors de l'explication théorique de la méthode et lors de l'explication du fonctionnement de l'algorithme, il existe plusieurs paramètres qu'il est nécessaire de bien dimensionner. En effet, leurs valeurs ont une grande importance à la fois sur le taux de réussite de l'algorithme, mais aussi sur ses performances, c'est-à-dire la vitesse à laquelle il va trouver des solutions.

Dans l'algorithme présenté, les différents paramètres sont :

- Le nombre d'itérations maximum avant l'arrêt de l'algorithme.
- Le nombre d'itérations d'exclusion d'une variable lorsqu'elle est marquée « Tabou »
- Le seuil pour la remise à zéro partielle du carré, c'est-à-dire le nombre de variables qui doivent être marquées « Tabou » pour procéder à la remise à zéro partielle du carré.
- Le nombre de variables à ré-initialiser lors d'une remise à zéro partielle.

IV.1 L'influence des différents paramètres sur l'algorithme

Avant de pouvoir correctement dimensionner les paramètres, il est important de bien comprendre l'influence de chaque paramètre sur l'algorithme et donc, de quelle manière la variation de leur valeur influera sur ses performances.

IV.1.1 L'influence du nombre maximum d'itérations

C'est le paramètre dont l'influence est la plus facile à déterminer. En effet, il ne sert qu'à garantir le caractère fini de l'algorithme, même si aucun carré magique n'est trouvé. Plus le nombre maximum d'itérations est grand et plus la probabilité de trouver une solution augmente. Par contre, le temps d'attente sera allongé si la recherche ne trouve pas de solution. A contrario, si le nombre d'itérations maximum est trop petit, la recherche risque de s'arrêter trop souvent avant de trouver une solution.

IV.1.2 L'influence du nombre d'itérations d'exclusion lors du marquage « Tabou »

Pour rappel, une variable est marquée « Tabou » lorsqu'elle a été choisie pour être permutée parce que son coût était le plus élevé mais qu'aucune configuration voisine ne permet de diminuer le coût global.

Dans ce cas de figure, cela signifie que dans la configuration actuelle, il n'y a aucune meilleure place pour la variable, et donc, qu'elle a un coût minimum pour la configuration actuelle. On suppose donc que la variable a une valeur correcte, c'est-à-dire permettant de conduire la fonction de coût vers un optimum. Elle est donc exclue de la recherche pendant quelques itérations, ce qui va permettre d'intensifier la recherche autour des configurations dans lesquelles cette variable a toujours cette valeur.

Si le nombre d'itérations d'exclusion est augmenté de façon excessive, on considérera les valeurs pour les variables « Tabou » comme bonnes trop longtemps. Cela augmente significativement le risque de se retrouver piégé dans des minima locaux en ne cherchant des solutions optimales que sur un nombre restreint de variables.

Si le nombre d'itérations d'exclusion est diminué excessivement, le cas inverse se produira. Les variables ne seront pas fixées suffisamment sur des valeurs supposées

bonnes, ce qui va biaiser le système d'intensification de la recherche, et donc nuire aux bonnes performances de la recherche.

IV.1.3 L'influence du seuil de remise à zéro partielle.

Ce paramètre va agir sur un des autres mécanismes indispensables à toute technique de recherche locale : le mécanisme de diversification. La diversification permet à la recherche d'explorer des configurations par lesquelles elle ne serait pas passée en respectant les règles établies par le principe général. Dans notre cas, ce mécanisme est mis en place par des remises à zéro partielles, qui sont dimensionnées au travers de deux paramètres ; l'un d'eux est le seuil, c'est-à-dire le nombre de variables devant être marquées « Tabou » pour que la procédure de remise à zéro soit déclenchée.

Si le seuil est trop bas, la recherche sera rendue fort aléatoire puisque dès que quelques variables seront marquées « Tabou », une remise à zéro partielle sera effectuée, laquelle est totalement aléatoire.

Par contre, si le seuil est trop haut, il faudra attendre qu'il y ait un grand nombre de variables marquées « Tabou » avant de procéder à une remise à zéro partielle. Il va ainsi falloir attendre de nombreuses itérations avant de détecter que la recherche a atteint un minimum local. L'influence dans ce sens est beaucoup plus difficile à cerner mais elle sera davantage analysée dans le chapitre traitant des essais faits sur différentes valeurs de ce paramètre.

IV.1.4 L'influence du nombre de variables ré-initialisées

Le nombre de variables ré-initialisées est le second paramètre permettant de dimensionner les remises à zéro partielles. Il va déterminer le nombre de variables choisies aléatoirement et qui seront mélangées.

L'effet de ce paramètre est simple à comprendre. Trop bas, le mécanisme de diversification sera atténué puisque très peu de variables seront mélangées. Il sera alors difficile de s 'éloigner des minima locaux. Trop élevé, la recherche est rendue beaucoup plus aléatoire, ce qui ne signifie pas pour autant qu'elle soit moins efficace.

IV.2 L'étude empirique de l'influence des paramètres et leur dimensionnement

L'étude théorique nous a permis de cerner l'influence de chaque paramètre sur la recherche mais, pour pouvoir correctement les analyser et les paramétrer, il faut procéder à une étude empirique. L'étude théorique permettra d'interpréter les résultats fournis par les jeux de test.

Le dimensionnement des paramètres va être fait au travers de la réalisations de jeux de tests pour lesquelles les valeurs de certaines variables vont être relevées en cours de recherche.

Les variables qui permettent d'identifier au mieux la façon dont la recherche a été effectuée sont les suivants :

- Le nombre d'itérations effectuées lors de la recherche.
- Le nombre de permutations effectuées, c'est-à-dire le nombre de fois où la recherche est passée d'une configuration à une voisine de coût inférieur.
- Le nombre de fois où une des variables du carré a été marquée « Tabou »
- Le nombre de fois où une remise à zéro partielle a été effectuée.
- Le temps total d'exécution
- La découverte d'un carré magique ou pas.

IV.2.1 Premières constatations

Les jeux de tests suivants ont été effectués avec les paramètres suivants sur des carrés d'ordre 9 et d'ordre 15:

- Nombre d'itérations maximum : 1000000
- Nombre d'itérations d'exclusion : n – 1 où n est l'ordre du carré
- Seuil de remise à zéro : 15% de n^2 où n^2 représente le nombre de variables du carré
- Nombre de variables remises à 0 : 10% de n^2 où n^2 représente le nombre de variables du carré

Tableau IV-1 : Les premiers jeux de test

Ordre	Nombre d'itérations	Remises à zéro	Permutations	Marquage « Tabou »	Temps écoulé [s]	Succès	Permutations / 100 Itérations	Remise à zéro / 100 itérations	Temps d'exécution [s] de 1000 itérations
9	10383	346	5542	4841	0.255	Oui	53.38	3.33	0.0245
9	71994	2405	38271	33723	1.764	Oui	53.16	3.34	0.0245
9	83535	2782	44346	39189	2.042	Oui	53.09	3.33	0.0244
9	32333	1096	17203	15130	0.794	Oui	53.21	3.39	0.0245
9	1000000	33464	532521	467479	24.523	Non	53.25	3.34	0.0245
9	115458	3864	61497	53961	2.826	Oui	53.26	3.35	0.0244
9	92563	3104	49250	43313	2.351	Oui	53.21	3.35	0.0253
9	222556	7513	118354	104202	5.46	Oui	53.18	3.38	0.0245
9	36304	1219	19300	17004	0.888	Oui	53.16	3.36	0.0244
9	227237	7662	120296	106941	5.552	Oui	52.94	3.37	0.0244
15	268652	3185	125924	142278	21.144	Oui	46.87	1.19	0.0787
15	333925	3957	156600	177325	26.242	Oui	46.90	1.18	0.0785
15	35414	419	16504	18910	2.775	Oui	46.60	1.18	0.0783
15	143817	1704	67457	76360	11.31	Oui	46.90	1.18	0.0786
15	13219	155	6193	7026	1.04	Oui	46.85	1.17	0.0786
15	787190	9324	368722	418468	61.746	Oui	46.84	1.18	0.0784
15	315592	3742	147733	167859	24.72	Oui	46.81	1.19	0.0783
15	289554	3430	135698	153856	22.736	Oui	46.86	1.18	0.0785
15	1000000	11822	468523	531477	78.431	Non	46.85	1.18	0.0784
15	756082	8964	354144	401938	59.274	Oui	46.84	1.19	0.0783

La première constatation concerne le pourcentage de permutations par itération et le pourcentage de remises à zéro par itération : ils restent identiques, pour un ordre de grandeur donné, pour autant que les paramètres ne changent pas. Cela signifie que la façon dont la recherche se déroule est bien fixée par les paramètres et que, toute aléatoire qu'elle soit, la recherche est bien guidée par les différents mécanismes mis en place.

Notons également que le nombre de marquage « Tabou » pour 100 itérations n'est pas repris dans le tableau. En effet, en observant l'algorithme de la recherche, il est facile de constater qu'à chaque itération, l'algorithme procède, soit une permutation, soit un marquage « Tabou ».

En observant les temps d'exécution par itération, on constate que l'ordre de grandeur réel de l'algorithme est effectivement légèrement meilleur que ne le laisse penser la complexité théorique. En effet, le temps d'exécution de 1000 itérations à l'ordre 9 est en

moyenne de 0.024 secondes. En se basant sur la complexité théorique, soit $O(n^3)$, le calcul du temps moyen d'exécution de 1000 itérations à l'ordre 15 donne $0.024 * (15/9)^3 = 0.111$, ce qui est supérieur au temps d'exécution obtenu, soit 0.0785 secondes de moyenne.

IV.2.2 L'effet de la variation du seuil de remise à zéro partielle

Pour analyser plus en profondeur l'influence de ce paramètre, on va observer le fonctionnement de la recherche avec les paramètres suivants :

- Ordre du carré : 9
- Nombre maximum d'itération : 1000000
- Nombre d'itérations d'exclusion : 35
- **Seuil de remise à zéro partielle : 81**
- Nombre de variables à remettre à zéro : 8

L'ordre du carré sélectionné pour ces essais est 9, qui est suffisamment grand pour être représentatif tout en permettant des temps de traitement relativement courts. A ce stade, la plupart des paramètres sont choisis relativement aléatoirement. Le seuil de remise à zéro est fixé à 81, c'est-à-dire que toutes les variables doivent être marquées « Tabou » pour qu'il y ait une remise à zéro.

Pour permettre d'observer la recherche, l'algorithme est lancé et, à chaque itération, le nombre de variables marquées « Tabou », les permutations ou marquages « Tabou » ainsi que les remises à zéro ou pas sont notées.

Le résultat est présenté ci-dessous sous forme de graphique pour les 1000 premières itérations sachant que le profil de la recherche reste le même pour les autres itérations.

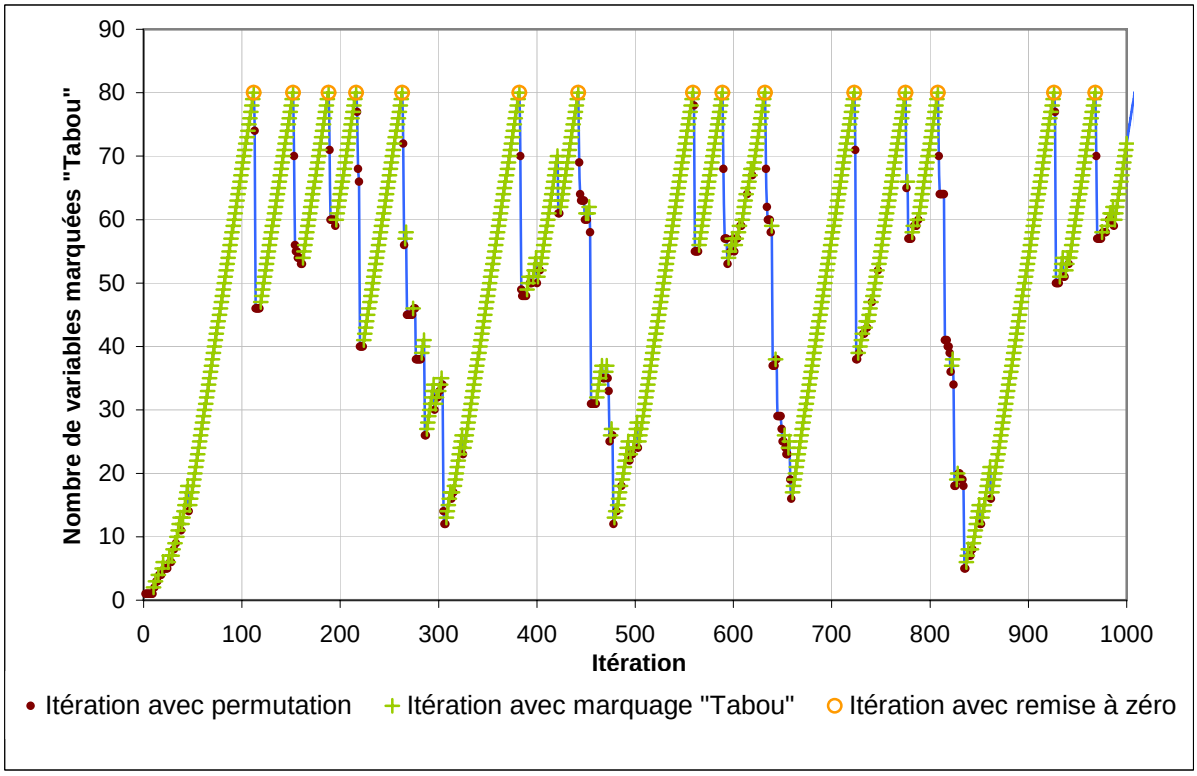


Figure IV-1 : Influence du seuil de remise à zéro (seuil = toutes les variables marquées "Tabou")

Nous pouvons y voir la courbe montrant le nombre de variables marquées « Tabou » en fonction du numéro d'itération. De plus, pour chaque itération, le type du point indique s'il y a eu permutation ou marquage « Tabou ».

La première constatation concerne le nombre de permutations qui est très inférieur au nombre de marquage « Tabou », mais en affinant un peu l'observation, on remarque qu'avant chaque remise à zéro partielle, il y a une longue suite de marquage « Tabou » sans permutation. Cela signifie que l'algorithme est momentanément piégé dans un minimum local. Or, étant donné qu'il n'y a plus de permutation, aucune variable « Tabou » ne peut-être remise en jeu puisque le nombre d'itérations d'exclusion n'est décrémenté qu'après une permutation. La recherche reste donc bloquée en attendant une remise à zéro partielle qui permettra de la relancer.

Le seuil a donc été choisi trop haut. Il est en effet inutile d'attendre que toutes les variables soient marquées « Tabou » pour procéder à une remise à zéro partielle. Par contre, même si le seuil trop élevé implique de nombreuses itérations inutiles, il n'empêche en aucun cas la réussite de la recherche. Il se marque uniquement par un plus grand nombre d'itérations.

Voyons maintenant l'effet contraire. Voici le graphe présentant le profil de la recherche avec les mêmes paramètres mais avec un seuil de remise à zéro fixé à 4.

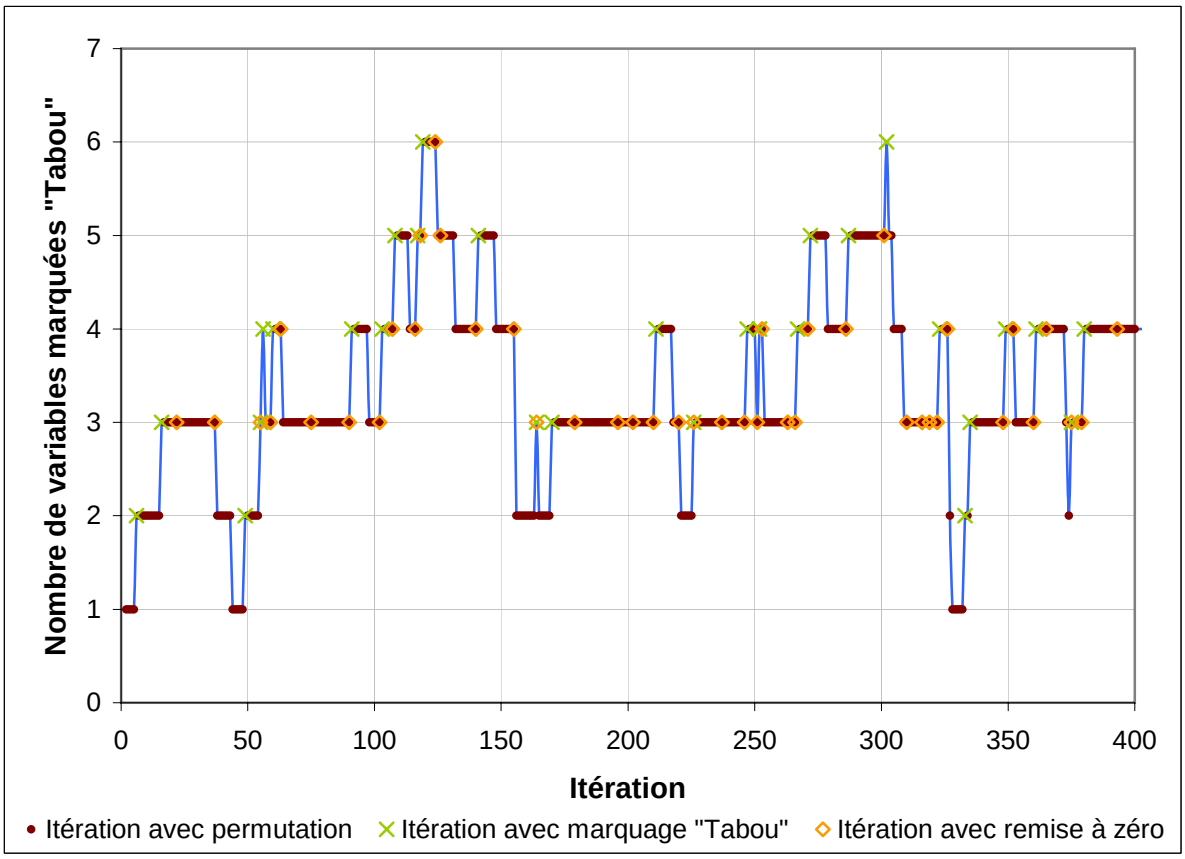


Figure IV-2 : Influence du seuil de remise à zéro (seuil = très peu de variables marquées "Tabou")

Dans ce cas, le nombre de remises à zéro est bien sûr beaucoup plus élevé, mais, la plupart s'avèrent inutiles, certaines ne permettant même pas de faire diminuer le nombre de

variables marquées « Tabou ». Il n'est bien sûr plus question de se perdre dans des minima locaux puisque la recherche est remise à zéro tout le temps et le nombre d'itérations avec marquage « Tabou » est très faible. Avec un seuil de remise à zéro aussi bas, le mécanisme d'intensification de la recherche est totalement évincé au profit d'une très grande diversification.

Alors qu'un seuil de remise à zéro trop élevé ne nuisait pas au taux de réussite de la recherche, un seuil aussi bas réduit fortement ce taux de réussite comme le montre le tableau ci-dessous.

Tableau IV-2 : Taux de réussite en fonction du seuil de remise à zéro

Seuil de remise à zéro	% de réussite
4	40%
8	90%
16	100%
24	100%
32	100%
40	100%
48	100%
54	80%
62	90%
81	90%

Au vu de ces différentes constatations, et après de nombreux essais en observant la montée vers le premier pic, une « bonne » valeur de seuil peut être de l'ordre de 10 à 20% du nombre de variables. Si l'on considère un seuil de 16, c'est-à-dire 20% de n^2, le graphique devient :

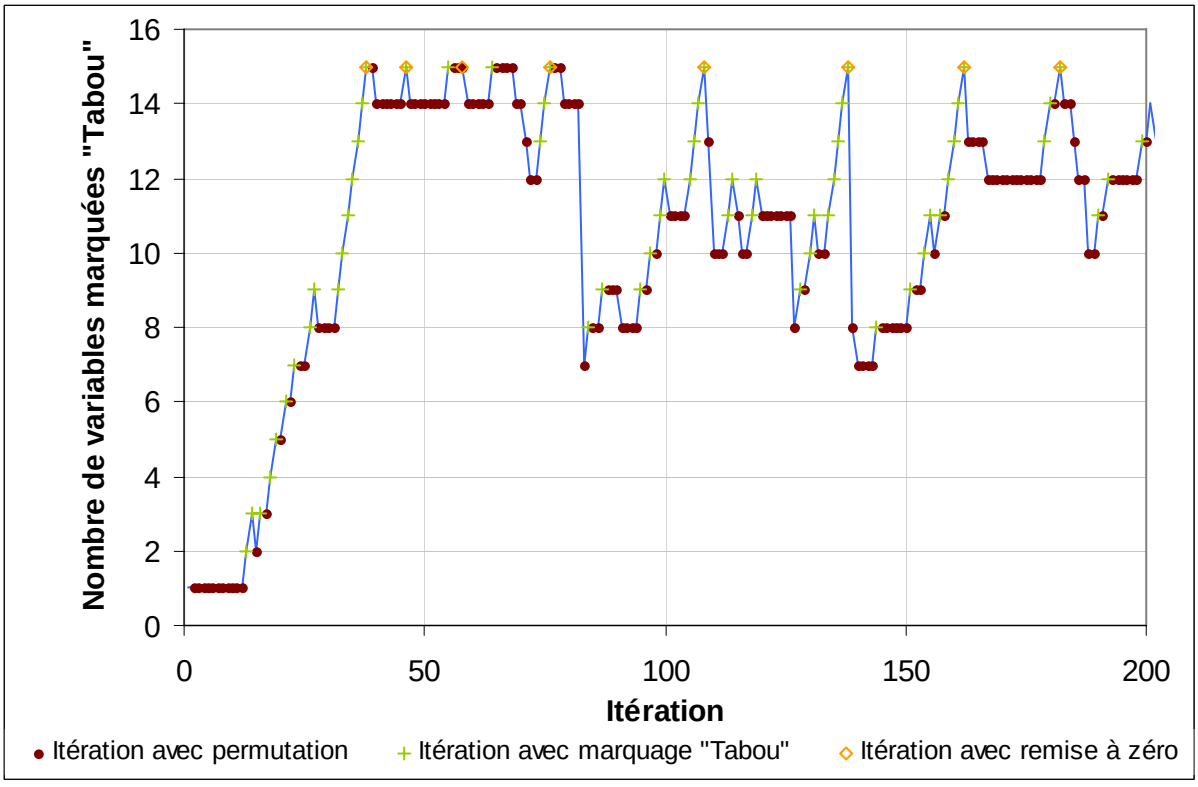


Figure IV-3 : Graphique présentant un seuil de remise à zéro bien dimensionné (20% de n^2)

IV.2.3 L'influence du nombre de variables à remettre à zéro

Au vu des conclusions présentées précédemment, le seuil de remise à zéro a été correctement dimensionné. Par contre, il est à noter que le nombre de variables marquées « Tabou » chute peu, voir pas du tout après une remise à zéro partielle. Cela est dû au faible nombre de variables qui sont remises à zéro. Comme indiqué lors de l'étude théorique de l'influence des paramètres, cela risque de faire tourner la recherche trop longtemps autour des mêmes valeurs, on peut d'ailleurs observer le phénomène dans le graphique précédent (Figure IV-3), notamment entre les itérations 40 et 80.

Voyons donc le profil de recherche suite à l'augmentation du nombre de variables remises à zéro. L'essai a été fait avec plusieurs valeurs, voici les graphiques pour deux d'entre eux : le premier avec 80 variables remises à zéro et le suivant avec 20. Les autres paramètres de la recherche n'ont pas été modifiés. Ils restent :

- Ordre du carré : 9
- Nombre maximum d'itération : 1000000
- Nombre d'itérations d'exclusion : 35
- Seuil de remise à zéro partielle : 16 (20% de n^2)

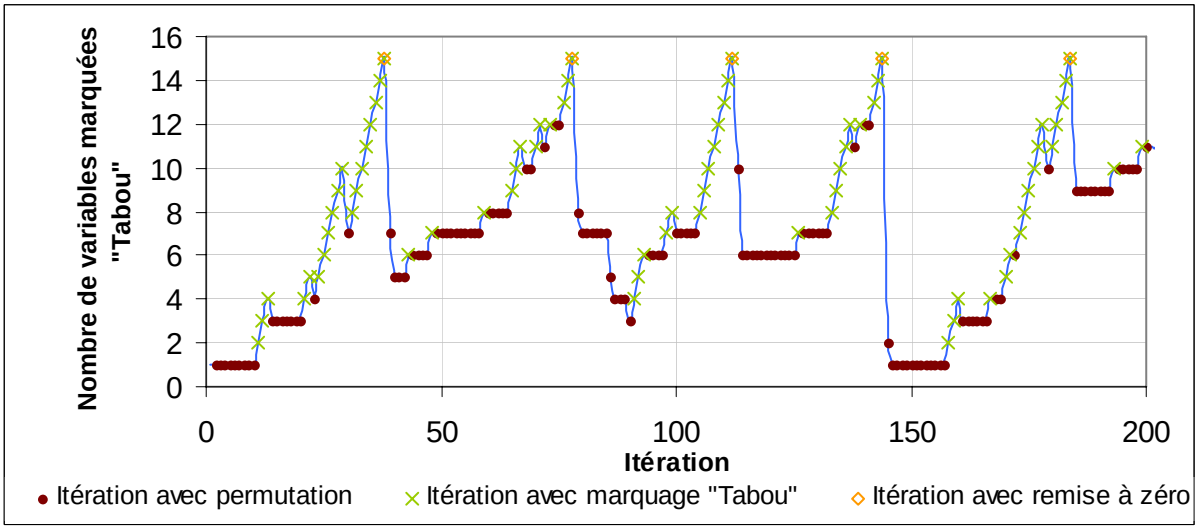


Figure IV-4 : Graphique présentant un profil de recherche avec une remise à zéro complète

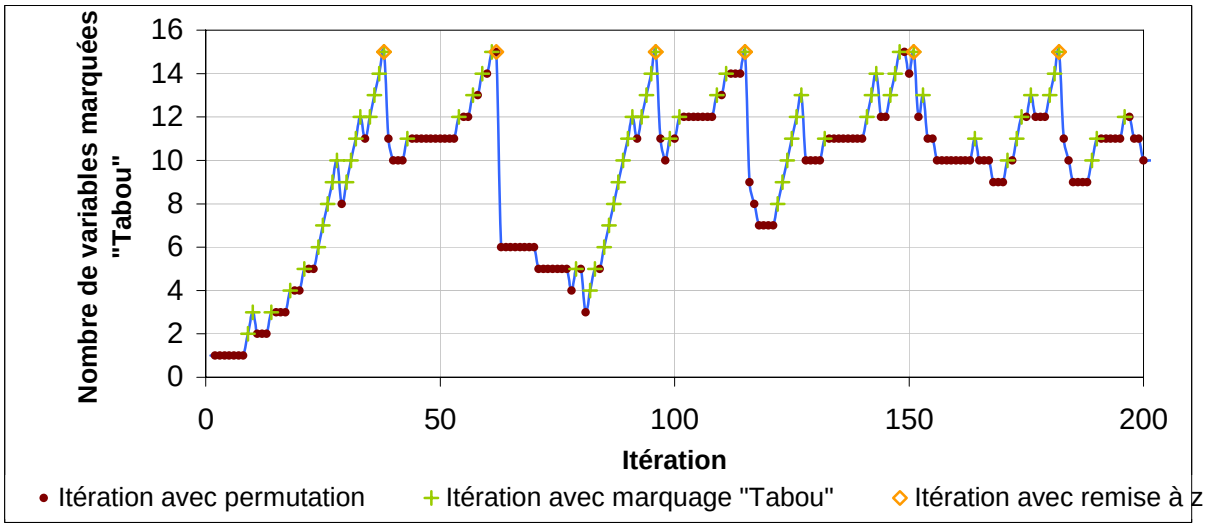


Figure IV-5 : Graphique présentant un profil de recherche avec une remise à zéro de 25% des variables

Comme le montrent les graphiques, l'augmentation du nombre de variables remises à zéro permet bien de diminuer le nombre de variables marquées « Tabou » après une remise à zéro, ce qui permet de varier les valeurs sur lesquelles la recherche se déroule.

Il est à remarquer que si le nombre de variables marquées « Tabou » après une remise à zéro partielle est plus faible lors d'une remise à zéro complète, le profil de la recherche reste le même pour autant que le nombre de variables remises à zéro soit suffisant. Ceci est illustré dans le tableau ci-dessous :

Tableau IV-3 : Effet de la variation du nombre de remises à zéro

Nombre de variables remises à zéro	Permutations sur 100 itérations	Remises à zéro par 100 itérations
4	64.07	7.99
8	64.52	5.02
16	65.33	3.62
24	63.71	3.49
32	64.63	3.14
40	64.11	3.04
48	62.39	3.25
56	62.89	3.05
64	62.16	3.13
72	62.48	2.87
81	61.36	2.98

Le tableau (Tableau IV-3) permet bien de constater que le nombre de remises à zéro est effectivement plus élevé si le nombre de variables remises à zéro est très faible, mais qu'il diminue rapidement pour se stabiliser lorsque le nombre de variables remises à zéro vaut à peu près 20% du nombre de variables du carré, c'est-à-dire le seuil à partir duquel la remise à zéro est effectuée. Ce paramètre a d'autre part peu d'influence sur la proportion de permutations par rapport au marquage « Tabou ».

Pour conclure, il semble qu'une bonne valeur à donner au nombre de variables remises à zéro soit un nombre légèrement supérieur au seuil de remise à zéro. Cela permet de remettre suffisamment de variables à zéro sans pour autant re-mélanger totalement le carré. L'augmenter un peu peut être intéressant si l'on désire augmenter l'effet stochastique de la recherche sans pour autant nuire aux performances.

IV.2.4 L'influence du nombre d'itérations d'exclusion

Pour bien analyser l'influence de ce paramètre, il faut étudier le dernier graphique obtenu (Figure IV-5). On y observe que le nombre de variables marquées « Tabou » diminue très peu en dehors des remises à zéro. Cela est dû au fait que le nombre d'itérations d'exclusion a été choisi trop grand. En effet, il est de 35, or on constate sur le graphique que le nombre d'itérations entre deux remises à zéro est, en moyenne, situé à peu près aux mêmes valeurs. Cela signifie qu'une variable marquée « Tabou » a autant de chance d'être remise en jeu par une remise à zéro que par la fin de son exclusion. Le phénomène sera évidemment d'autant plus marqué que l'on augmentera le nombre d'itérations d'exclusion.

Si on diminue suffisamment le nombre d'itérations d'exclusion d'une variable marquée « Tabou » par rapport au nombre d'itérations entre deux remises à zéro, on obtient alors le graphique suivant pour lequel il est diminué à 9, soit la valeur de l'ordre n.

Figure IV-6 : Graphique présentant une réduction du nombre d'itérations d'exclusion

Le nombre de remises à zéro est alors diminué et le nombre de variables marquées « Tabou » diminue autrement que par la procédure de remise à zéro, ce qui est beaucoup plus correct vis-à-vis de la logique de la recherche.

Il faut cependant faire attention à ne pas trop diminuer le nombre d'itérations d'exclusion. En effet, le profil engendré par 2 itérations d'exclusion est présenté ci-dessous :

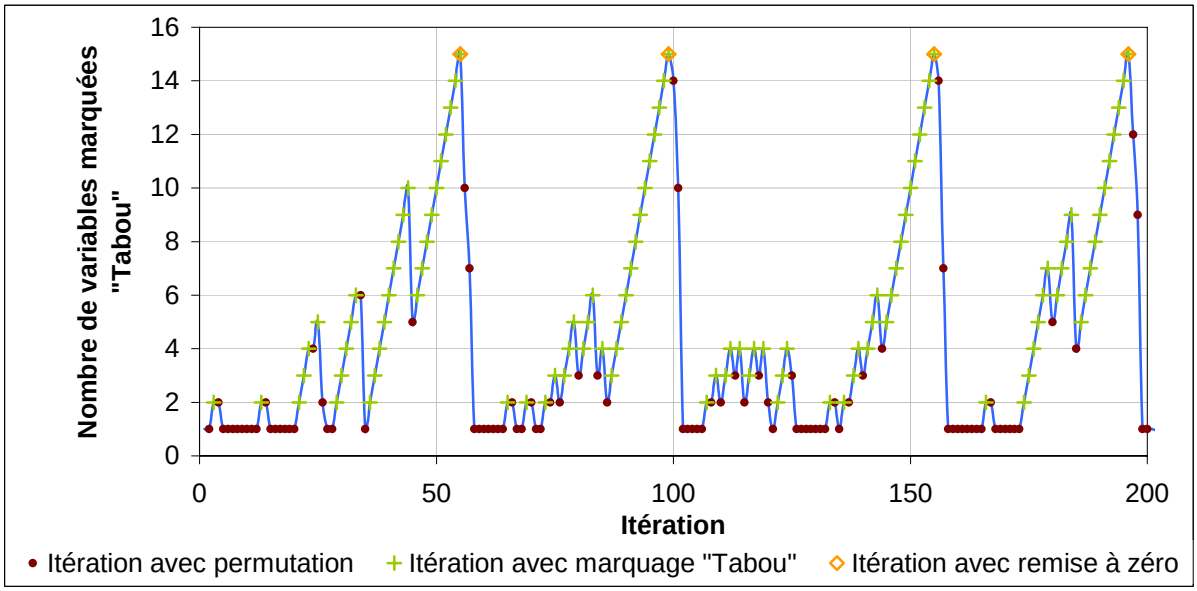


Figure IV-7 : Graphique présentant un nombre d'itérations d'exclusion trop faible

Le nombre de variables marquées « Tabou » reste presque toujours nul, égal à 1 dans notre cas car la case centrale du carré est toujours « Tabou », ce qui induit la disparition quasi totale du mécanisme d'intensification.

De plus, on constate la réapparition des plus nombreux marquages « Tabou » avant de détecter ce qui est considéré comme un minimum local. Cela montre à quel point les valeurs correctes pour les différents paramètres sont également fortement dépendantes des valeurs données aux autres paramètres.

IV.2.5 Le nombre maximum d'itérations

L'influence de ce paramètre est beaucoup plus simple. S'il est trop bas, la recherche risque d'être arrêtée trop tôt, ce qui va en diminuer son taux de réussite. L'augmentation, quant à elle, ne nuit jamais au taux de réussite. Par contre, une forte augmentation du nombre maximum d'itérations demandera une plus grande patience en cas d'échec de la recherche.

Pour l'ordre 9, un nombre d'itérations entre 250000 et 500000 semble très largement suffisant.

IV.2.6 Conclusions pour l'ordre 9

Après de nombreux tests, il semble que les paramètres suivants permettent un bon fonctionnement pour une recherche sur l'ordre 9 :

- Nombre maximum d'itérations : 250000
- Nombre d'itérations d'exclusion : 9 (n)
- Nombre de variables remises à zéro : 20 (25% de n^2)
- Seuil de remise à zéro : 16 (20%de n^2)

Le profil de recherche correspondant est celui de la fFigure IV-6, tandis que le tableau ci-dessous présente un aperçu des résultats obtenus avec ces paramètres :

Tableau IV-4 : Tableau des performances avec les paramètres correctement réglés

Nombre d'itérations	Nombre de remises à zéro	Nombre de permutations	Nombre de marquages « Tabou »	Temps d'exécution [s]	Trouvé
54597	1363	28523	26074	1.288	Oui
178310	4464	93216	85094	4.174	Oui
50858	1267	26028	24830	1.194	Oui
53854	1339	28078	25776	1.271	Oui
1363	31	689	674	0.034	Oui
5531	133	2692	2839	0.133	Oui
38905	960	20479	18426	0.915	Oui
19597	482	9824	9773	0.458	Oui
63830	1617	33068	30762	1.501	Oui
44116	1090	22628	21488	1.035	Oui
Moyenne 51096.1	1274.6	26522.5	24573.6	1.200	

Les performances présentées sont les meilleures obtenues jusqu'alors avec cet ordre de grandeur. Le taux de réussite de la recherche se situe entre 95 et 99%.

IV.2.7 La transposition des paramètres trouvés dans les ordres de grandeurs plus élevés

La transposition des paramètres dans l'ordre 15 donne :

- Nombre maximum d'itérations : 1000000
- Nombre d'itérations d'exclusion : 15 (n)
- Nombre de variables remises à zéro : 56 (25% de n^2)
- Seuil de remise à zéro : 45 20% de n^2)

Ces paramètres provoquent le profil de recherche repris dans le graphique suivant :

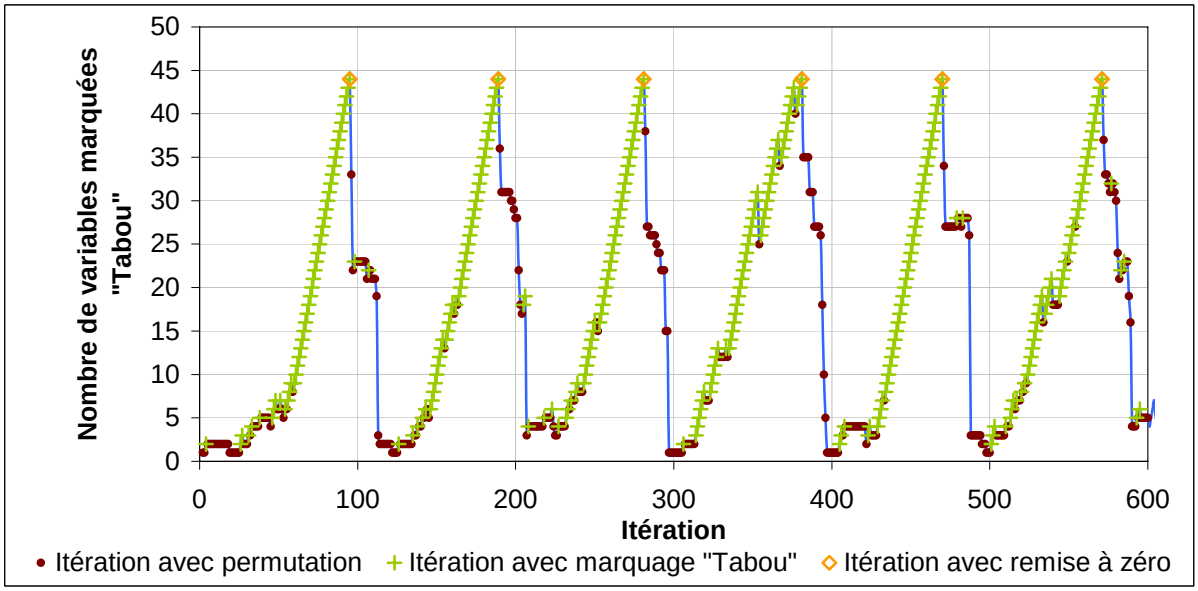


Figure IV-8 : Profil de recherche dans l'ordre 15

On constate une légère dégradation du profil de la recherche dans le sens où l'on doit faire quelques itérations supplémentaires avant de détecter un minimum local. Néanmoins, les performances de l'algorithme se dégradent fortement en terme de taux de réussite si le seuil de remise à zéro est diminué.

Toujours avec ces mêmes paramètres cités, les performances de l'algorithme sont les suivantes :

Tableau IV-5 : Performances de la recherche pour l'ordre 15

	Nombre d'itérations	Nombre de remises à zéro	Nombre de permutations	Nombre de marquage "Tabou"	Temps d'exécution [s]	Succès
	367698	3741	161144	206554	28.20	Oui
	515514	5358	230690	284824	39.75	Oui
	34233	360	15451	18782	2.63	Oui
	143854	1458	62517	81337	11.06	Oui
	836705	8711	373469	463236	64.36	Oui
	229700	2358	101477	128223	17.66	Oui
	431614	4429	190173	241441	33.10	Oui
	185986	1848	79500	106486	14.23	Oui
	151411	1563	67066	84345	11.64	Oui
	425266	4426	190395	234871	32.69	Oui
Moyenne	332198.1	3425.2	147188.2	185009.9	25.53	

Comme le montre le tableau 5, les performances sont moindres pour l'ordre 15. En effet, le temps d'exécution est plus long, ce qui est normal étant donné la complexité de l'algorithme ($O(n^3)$). Le nombre d'itérations nécessaire pour générer un carré magique a également augmenté, ce qui se justifie par l'augmentation du nombre de variables. Néanmoins, le taux de réussite de la recherche reste très élevé, il se situe entre 90 et 95%.

La transposition des paramètres dans l'ordre 20 donne ceci :

- Nombre maximum d'itérations : 3000000
- Nombre d'itérations d'exclusion : 20 (n)
- Nombre de variables remises à zéro : 100 (25% de n^2)
- Seuil de remise à zéro : 80 (20% de n^2)

La recherche s'exécute alors avec les performances suivantes :

Tableau IV-6 : Performances de la recherche pour l'ordre 20

Nombre d'itérations	Nombre de remises à zéro	Nombre de permutations	Nombre de marquage "Tabou"	Temps d'exécution [s]	Succès
477266	3160	196352	280914	80.01	Oui
1273622	8422	523123	750499	213.62	Oui
1965187	12971	806391	1158796	329.34	Oui
449312	2968	184397	264915	75.30	Oui
217410	1422	88545	128865	36.45	Oui
3000052	19788	1230196	1769856	503.81	Non
148418	984	61163	87255	24.87	Oui
1399970	9224	572253	827717	234.74	Oui
2374696	15670	974095	1400601	398.93	Oui
1719963	11350	705025	1014938	288.44	Oui
Moyenne 1302589.6	8595.9	534154	768435.6	218.55	

Ce tableau montre bien que les phénomènes constatés lors des essais sur l'ordre 15 continuent et s'amplifient légèrement sur l'ordre 20. En effet, le nombre d'itérations avec marquage « Tabou » augmente par rapport au nombre d'itérations avec permutations. Cette augmentation étant toujours due en grande partie à l'augmentation du nombre d'itérations nécessaire à la détection d'un minimum local.

IV.2.8 Conclusions

Il semble que les paramètres déterminés fonctionnent bien mais avec une légère dégradation au fur et à mesure que l'ordre de grandeur augmente. Etant donné que le travail ne portait pas sur les ordres de grandeurs supérieurs à 23, ils sont donc suffisants. Néanmoins, pour une recherche sur des ordres de grandeurs plus élevés, il serait intéressant de dimensionner une nouvelle fois ces paramètres.

Une autre solution envisageable est de modifier les paramètres de la recherche en cours de travail. En effet, maintenant que les liens entre les différents paramètres sont bien connus, il est tout à fait possible de les surveiller au cours des itérations et de les modifier en conséquence. Néanmoins, tous ces contrôles sur les paramètres alourdissent l'algorithme. De plus, il faudrait quantifier et valider précisément ces contrôles sans quoi les recherches pourraient se perdre. Cela demanderait donc un travail important, pour de faibles améliorations étant donné les ordres de grandeurs sur lesquels le travail porte. L'expérience n'a donc pas été menée plus loin.

Chapitre V
CONCLUSION

V.1 Objectifs rencontrés

Les objectifs mentionnés dans le chapitre I sont atteints. En effet, dans le cadre du travail présenté, (ordre de grandeur compris entre 6 et 23), la recherche aboutit dans plus de 90% des cas. Pour les ordres de grandeurs inférieurs à 10, le taux de réussite est même proche des 100%

En ce qui concerne les performances des temps de réponse, il est beaucoup plus difficile de juger. En effet, l'idéal eut été de pouvoir les comparer avec d'autres méthodes de résolution. Malheureusement, il ne m'était matériellement pas possible d'implémenter, et surtout de tester et de dimensionner d'autres méthodes pour les comparer. Néanmoins, d'un point de vue pratique, la vitesse de l'algorithme semble suffisante. En effet, le temps de réponse moyen est de 1.2 secondes pour l'ordre 9, de 25 secondes pour l'ordre 15 et de 300 secondes pour l'ordre 20. Bien entendu, la partie aléatoire de la recherche reste prépondérante, ce qui ne simplifie pas les statistiques.

Voici quelques avantages particuliers de l'algorithme implémenté :

- Il est totalement indépendant de l'ordre dans lequel on veut travailler. La limite ne se situe théoriquement que dans la taille des tableaux.
- Il permet de créer des carrés magiques réguliers, c'est-à-dire contenant les nombres de 1 à N^2, mais il permet également de travailler dans d'autres ensembles. L'algorithme présenté ici permet de générer des carrés magiques, après de légères adaptations, directement dans l'ensemble de 0 à N^2-1 dans lequel il est parfois plus facile de travailler pour certains calculs. Il suffit en effet de modifier la génération du cas de base pour qu'il contienne les éléments de 0 à N^2-1 et de changer le calcul de la constante magique qui, dans cet ensemble, est de $n * (n^2 - 1) / 2$. Cette possibilité a été testée et fonctionne parfaitement. De manière plus générale, l'algorithme peut fonctionner pour n'importe quel ensemble de nombres pour autant que l'on puisse générer un cas de base et calculer la constante magique correspondante.
- Autre avantage de l'algorithme implémenté : la gestion des éléments fixés. Pour ce travail, seuls les éléments du noyau étaient fixés, mais l'algorithme est implémenté de telle manière qu'il est facile de choisir d'autres éléments que l'on voudrait fixer. Il suffit pour cela d'initialiser la liste des éléments marqués « Tabou » en conséquence. On peut même imaginer que certains éléments soient marqués « Tabou » pendant les X premières itérations et qu'ils soient remis dans la recherche pour les suivantes si la recherche n'a pas encore abouti.

V.2 Améliorations et perspectives

En terme de performances de l'algorithme, le temps d'exécution des itérations me paraît difficilement améliorable. En effet, la partie la plus longue de chaque itération étant l'exploration du voisinage, il me paraît difficile de trouver une méthode permettant de déterminer la meilleure permutation sans toutes les essayer.

Afin de gagner en performances, notamment pour les ordres de grandeur les plus élevés, il serait intéressant de modifier les paramètres de la recherche de manière dynamique. Dans l'algorithme présenté, ils ne sont déterminés que statiquement, sur base empirique. L'idée serait de contrôler la pertinence des valeurs des paramètres en cours de recherche et de les adapter le cas échéant. Une possibilité, par exemple, serait de contrôler que le nombre d'itérations d'exclusion est inférieur au nombre d'itérations entre deux remises à zéro partielles.

Il serait également possible d'étendre la recherche aux degrés de magie supérieurs, c'est-à-dire de l'étendre à la création de carrés bi-magiques ou même, tri-magiques. En théorie, il suffit de modifier les contraintes et le calcul du coût sur ces contraintes, le reste de la méthode restant valable. A l'heure actuelle, aucun résultat valable n'a été obtenu avec des carrés bi-magiques.

Il serait sûrement intéressant d'étudier d'autres méta-heuristiques et des les adapter au problème des carrés magiques. Toutes ont leurs avantages et il est difficile de dire lesquelles donneront des résultats positifs sans les avoir testées.

Les carrés magiques sont un domaine très vaste, que j'estime n'avoir fait qu'effleurer avec ce travail. Le nombre d'outils que l'informatique peut y apporter est très important, notamment pour la recherche ou l'exploitation de carrés magiques présentant des caractéristiques particulières (Panmagie, degré de magie élevé, hémi-magie,…)

Bibliographie

[Aarts et al, 1990] Aarts E., Korts J., *Simulated Annealing and Boltzmann Machines. A stochastic Approach to Combinatorial Optimization and Neural Computing*, John Wiley, Chichester and New York, 1990

[Bouteloup, 1991] Bouteloup J., *Carrés Magiques, Carrés Latins et Eulériens*, Editions du Choix, Argenteuil, 1991

[Codognet, 2001] Codognet P. ,Diaz D., *Yet Another Local Search Method for Constraint Solving*, Stochastic Algorithms, Foundations and Applications (SAGA), Berlin, Germany, 2001.

[Descombes, 2000a] Descombes R., *Les carrés magiques. Histoire, théorie et technique du carré magique de l'Antiquité aux recherches actuelles*, Vuibert, Paris, 2000

[Descombes, 2000b] Descombes R., « Chapitre 19 : La construction des carrés magiques d'ordre impair », *Les carrés magiques. Histoire, théorie et technique du carré magique de l'Antiquité aux recherches actuelles*, Vuibert, Paris, pages 183-185, 2000

[Descombes, 2000c] Descombes R., « Chapitre 23 : La méthode de Ralph Strachey », *Les carrés magiques. Histoire, théorie et technique du carré magique de l'Antiquité aux recherches actuelles*, Vuibert, Paris, pages 183-185, 2000

[Dréo et al, 2005a] Dréo J., Pétrowski A., Siarry P., Taillard E., «La méthode du recuit simulé » in *Métaheuristiques pour l'optimisation difficile*, Eyrolles, Paris, pages 21-42, 2005

[Dréo et al, 2005b] Dréo J., Pétrowski A., Siarry P., Taillard E., «La recherche avec des tabous » in *Métaheuristiques pour l'optimisation difficile*, Eyrolles, Paris, pages 43-68, 2005

[Droesbeke, 1997] Droesbeke J-J, Tassi Ph., *Histoire de la statistique*, Presses Universitaires de Paris, Paris, 1997

[Gérardin, 1986] Gérardin L.: Les carrés magiques. Mystérieuses harmonies de nombre, Dangles, Paris, 1986

[Glover, 1997] Glover F., Laguna M., *Tabu Search*, Kluwer Academic Publishers, Norwell, Etats-Unis, 1997

[Jacques, 2006] Dr Jacques A., *fascicule1*, Charleroi, février 2006

[Leclercq, 2005] Leclercq J-P, « Optimisation combinatoire » in *Cours d'optimisation combinatoire et discrète*, Namur, 2005

ANNEXES

A.1 Code source du programme de génération de carrés magiques

```
#include <utility.h>
#include "toolbox.h"
#include <formatio.h>
#include <ansi_c.h>
#include "buildInterfaceOrdre9.h"

//    Génération de carrés magiques.
//    ------------------------------------------------

//    Définition des constantes.
//    -----------------------------------
#define ORDRE_CARRE             9           //    Ordre du carré magique.
#define NB_ITERATIONS           250000      //    Nombre d'itérations  maximum.
#define NB_ITER_EXCL            8           //    Nombre d'itérations d'exclusion.
#define NB_VARIABLES_RESET      20          //    Nombre de variables à remettre à zéro.
#define SEUIL_RESET_PARTIEL     16          //    Seuil de remise à zéro partielle.

//    Déclaration des variables globales.
//    -----------------------------------------------
long listeTabu[ORDRE_CARRE * ORDRE_CARRE];    //    Liste Tabu.
int nbVariablesTabou;                         //    Nombre de variables marquées "Tabou".
//    Variables utilisées pour le stockage du coût sur les contraintes.
int coutLignes[ORDRE_CARRE], coutColonnes[ORDRE_CARRE];

//    Déclaration des prototypes des fonctions.
//    -----------------------------------------
int fGenereCarre (int pDiag1[], int pDiag2[], int pInterface);
void genereCasBase (int pCarreBase [], int pOrdre, int pDiag1[], int pDiag2[]);
void calculCoutContraintes(int pOrdre, long pSomme, int pCarre[]);
void MAJCoutContraintes(int pOrdre, int indexSwap1, int valSwap1, int indexSwap2,
     int valSwap2);
void calculCoutVariables (int pCarre[], int pIndexCoutMax[], int *pCout, int *pNbCoutsMax,
     int pOrdre, int pSomme, int *pCoutGlobal);
void fCoutMinimum (int pCarre[], int pOrdre, int *pIndexCoutMin, int *nouveauCout,
     int indexVariableAPermuter, int pSomme, int pCoutGlobal);
void fCoutGlobal (int pCarre[], int pOrdre, int pSomme, int *pCout);
void fResetPartiel (int pCarre[], int pOrdre, int pNbVariables);
//    - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
void fInitTabu (void);
void fAddVariable (int pIndex);
void fMAJTabu (void);
//    - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
int fVerifValiditeDiag (int pDiag1[], int pDiag2[], int pOrdre);
//    - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
int fSauveCarre (int pInterface, int pOrdre);
//    ---------------------------------------------------------------------------------

//    Fonction principale.
//    ----------------------------
int main (int argc, char *argv[])
{
     int iPanel, iSortie, controle, iRetour;
     int diagonale1[ORDRE_CARRE], diagonale2[ORDRE_CARRE];

     //    Chargement et affichage de l'interface pour l'utilisateur.
     iPanel = BuildP_CARRE (0);
     if (iPanel < 0)    return -1;
     DisplayPanel (iPanel);

     //    Gestion de l'interface graphique.
     iSortie = 0;
     while (!iSortie)
     {
          GetUserEvent (1, 0, &controle);

          //    On quitte le programme.
          if (controle == P_CARRE_CB_QUITTER)
          {
               iSortie = 1;
          }
```

```
        //    On lance le traitement.
        if (controle == P_CARRE_CB_START)
        {
            //    Récupération des valeurs de diagonales demandées par l'utilisateur.
            GetCtrlVal (iPanel, P_CARRE_C0, &diagonale1[0]);
            GetCtrlVal (iPanel, P_CARRE_C10, &diagonale1[1]);
            GetCtrlVal (iPanel, P_CARRE_C20, &diagonale1[2]);
            GetCtrlVal (iPanel, P_CARRE_C30, &diagonale1[3]);
            GetCtrlVal (iPanel, P_CARRE_C40, &diagonale1[4]);
            GetCtrlVal (iPanel, P_CARRE_C50, &diagonale1[5]);
            GetCtrlVal (iPanel, P_CARRE_C60, &diagonale1[6]);
            GetCtrlVal (iPanel, P_CARRE_C70, &diagonale1[7]);
            GetCtrlVal (iPanel, P_CARRE_C80, &diagonale1[8]);
            GetCtrlVal (iPanel, P_CARRE_C8, &diagonale2[0]);
            GetCtrlVal (iPanel, P_CARRE_C16, &diagonale2[1]);
            GetCtrlVal (iPanel, P_CARRE_C24, &diagonale2[2]);
            GetCtrlVal (iPanel, P_CARRE_C32, &diagonale2[3]);
            GetCtrlVal (iPanel, P_CARRE_C40, &diagonale2[4]);
            GetCtrlVal (iPanel, P_CARRE_C48, &diagonale2[5]);
            GetCtrlVal (iPanel, P_CARRE_C56, &diagonale2[6]);
            GetCtrlVal (iPanel, P_CARRE_C64, &diagonale2[7]);
            GetCtrlVal (iPanel, P_CARRE_C72, &diagonale2[8]);

            //    On empêche les sauvegardes durant la recherche.
            SetCtrlAttribute (iPanel, P_CARRE_CB_SAUVER, ATTR_DIMMED, 1);


            //    Vérification de la validité des pDiagonales.
            iRetour = fVerifValiditeDiag (diagonale1, diagonale2, ORDRE_CARRE);
            if (iRetour)
            {
                //    tentative de génération d'un carré magique.
                iRetour = fGenereCarre (diagonale1, diagonale2, iPanel);
            }
            else
                MessagePopup ("Recherche non-effectuée",
                    "Les diagonales ne sont pas conformes");
        }

        if (controle == P_CARRE_CB_SAUVER)
        {
            //    Sauvegarde du carré dans un fichier ASCII.
            iRetour = fSauveCarre (iPanel, ORDRE_CARRE);
            if (iRetour == -1)
                MessagePopup ("Carré non-sauvegardé", "Erreur lors de la sauvegarde");
        }
    }

    //    Déchargement de l'interface et libération de la mémoire.
    DiscardPanel (iPanel);

    return 0;
}
//    ____________________________________________________________________________
```

```
//  ---------------------------------------------------------------------------------------------------------
//  fGenereCarre
//  Fonction permettant la recherche et la génération d'un carré magique.
//  Paramètres :
//      - pDiag1        : Tableau contenant les valeurs de la 1ère diagonale.
//      - pDiag2        : Tableau contenant les valeurs de la 2ème diagonale.
//      - pInterface : Numéro associé à l'interface lors de son chargement.
//  Valeur retournée : La fonction retourne 0 en cas d'échec ou 1 en cas de succès.
//  ---------------------------------------------------------------------------------------------------------
int fGenereCarre (int pDiag1[], int pDiag2[], int pInterface)
{
    int carreMagique[ORDRE_CARRE*ORDRE_CARRE];
    int i,j, swap;
    char afficheCarre[200];
    long nbIterations;

    int indexCoutMax, coutMax, coutGlobal;
    long somme;
    int indexCoutMin, coutMin, indexTmp;
    int varCoutsMax[ORDRE_CARRE*ORDRE_CARRE], nbCoutsMax;
    double resultatRandom, randomMax;
    time_t currentTime;

    int iRetour;

    //  Initialisation du générateur de nombres aléatoires.
    srand (time (&currentTime));

    //  Calcul de la somme magique pour un carré d'ordre ORDRE_CARRE.
    somme = ORDRE_CARRE * (ORDRE_CARRE * ORDRE_CARRE + 1) / 2;

    //  Initialisation du nombre d'itérations effectuées.
    nbIterations = 0;

    //  Génération d'un carré de base aléatoire.
    genereCasBase (carreMagique, ORDRE_CARRE, pDiag1, pDiag2);

    //  Initialisation de la liste Tabu.
    fInitTabu ();

    //  Calcul du coût sur les contraintes.
    calculCoutContraintes(ORDRE_CARRE, somme, carreMagique);

    //  Lancement de la partie itérative de l'algorithme.
    do
    {
        //  Définition du point de retour lors d'une remise à zéro partielle.
        resetPartiel:;

        //  Définition du point de retour lorsque l'on veut relancer un nouveau
        //      calcul du coût maximum des variables sans passer à l'itération suivante.
        nouveauCalculCoutVariables:;

        //  Calcul du coût global et des variables de coût maximum non marquées "Tabou".
        //  Les variables de coût maximum équivalent sont stockées dans un vecteur.
        calculCoutVariables (carreMagique, varCoutsMax, &coutMax, &nbCoutsMax,
            ORDRE_CARRE, somme, &coutGlobal);

        //  Si le carré généré n'est pas magique, on continue le traitement.
        //  Si le coût global est nul, le carré généré est magique.
        if (coutGlobal > 0)
        {
            //  Définition du point pour permettre une nouvelle recherche des coûts
            //      minimum à partir d'un nouveau coût max.  Mais sans qu'il soit
            //      nécessaire de refaire le calcul des coûts max.
            nouveauCoutMax:;

            //  Choix aléatoire d'une variable de coût max dans le vecteur des
            //      variables calculées.
            randomMax = nbCoutsMax - 1;
            resultatRandom = rand ();
            resultatRandom /= RAND_MAX;
            resultatRandom *= randomMax;
            indexTmp = RoundRealToNearestInteger (resultatRandom);
```

```
            indexCoutMax = varCoutsMax[indexTmp];

            //    Recherche de la configuration voisine avec coût minimum.
            fCoutMinimum (carreMagique, ORDRE_CARRE, &indexCoutMin, &coutMin,
                indexCoutMax, somme, coutGlobal);

            //    Si on a pas trouvé de permutations avec un coût global plus petit.
            if ( indexCoutMin == -1)
            {
                //    Mise à jour de la liste tabous pour exclure la variable des
                //          prochaines itérations.
                fAddVariable (indexCoutMax);

                //    Si le seuil du nombre de variables marquées "Tabou" est atteint,
                //          on procède à une remise à zéro partielle du carré.
                if (nbVariablesTabou >= SEUIL_RESET_PARTIEL)
                {
                    fResetPartiel (carreMagique, ORDRE_CARRE, NB_VARIABLES_RESET);
                    //    On passe directement à l'itération
                    nbIterations++;
                    goto resetPartiel;
                }

                //    Choix de la nouvelle variable de coût maximum.
                if (nbCoutsMax > 1)//    S'il reste des éléments dans le vecteur des coûts max.
                {
                    //    Suppression de la variable courante dans le vecteur des coûts max.
                    nbCoutsMax--;
                    for (i = indexCoutMax; i < nbCoutsMax - 1; i++)
                        varCoutsMax[i] = varCoutsMax[i + 1];
                    // On recommence la recherche du cout min. avec le nouveau coût max extrait.
                    nbIterations++;
                    goto nouveauCoutMax;
                }
                else    //    Plus d'éléments dans le vecteur --> Nouvelle recherche de coûts
                        //          max avec les valeurs non-Tabu restantes.
                {
                    nbIterations++;
                    goto nouveauCalculCoutVariables;
                }
            }
            else
            {
                //    on a trouvé une nouvelle config avec un coût plus petit
                //          --> Permutation des deux variables.
                swap                            = carreMagique[indexCoutMax];
                carreMagique[indexCoutMax]    = carreMagique[indexCoutMin];
                carreMagique[indexCoutMin]    = swap;

                //    Mise à jour des coûts des contraintes.
                MAJCoutContraintes (ORDRE_CARRE, indexCoutMin,
                    carreMagique[indexCoutMin], indexCoutMax,
                    carreMagique[indexCoutMax]);
            }

            //    Mise à jour de la liste tabu pour l'itération suivante.
            fMAJTabu ();

            //    Passage à l'itération suivante.
            nbIterations++;
            SetCtrlVal (pInterface, P_CARRE_NO_ITER, nbIterations);
        }
    //    On boucle tant que l'on a pas trouvé de solutions et que le nombre d'itérations
    //          est inférieur au nombre maximum d'itérations.
    }    while (nbIterations < NB_ITERATIONS && coutGlobal != 0);

    //    Affichage du carré si réussite ou message d'erreur en cas d'échec.
    if (coutGlobal == 0)
    {
        iRetour = 1;
        SetCtrlVal (pInterface, P_CARRE_C0, carreMagique[0]);
        SetCtrlVal (pInterface, P_CARRE_C1, carreMagique[1]);
        SetCtrlVal (pInterface, P_CARRE_C2, carreMagique[2]);
        SetCtrlVal (pInterface, P_CARRE_C3, carreMagique[3]);
```

```
SetCtrlVal (pInterface, P_CARRE_C4, carreMagique[4]);
SetCtrlVal (pInterface, P_CARRE_C5, carreMagique[5]);
SetCtrlVal (pInterface, P_CARRE_C6, carreMagique[6]);
SetCtrlVal (pInterface, P_CARRE_C7, carreMagique[7]);
SetCtrlVal (pInterface, P_CARRE_C8, carreMagique[8]);
SetCtrlVal (pInterface, P_CARRE_C9, carreMagique[9]);
SetCtrlVal (pInterface, P_CARRE_C10, carreMagique[10]);
SetCtrlVal (pInterface, P_CARRE_C11, carreMagique[11]);
SetCtrlVal (pInterface, P_CARRE_C12, carreMagique[12]);
SetCtrlVal (pInterface, P_CARRE_C13, carreMagique[13]);
SetCtrlVal (pInterface, P_CARRE_C14, carreMagique[14]);
SetCtrlVal (pInterface, P_CARRE_C15, carreMagique[15]);
SetCtrlVal (pInterface, P_CARRE_C16, carreMagique[16]);
SetCtrlVal (pInterface, P_CARRE_C17, carreMagique[17]);
SetCtrlVal (pInterface, P_CARRE_C18, carreMagique[18]);
SetCtrlVal (pInterface, P_CARRE_C19, carreMagique[19]);
SetCtrlVal (pInterface, P_CARRE_C20, carreMagique[20]);
SetCtrlVal (pInterface, P_CARRE_C21, carreMagique[21]);
SetCtrlVal (pInterface, P_CARRE_C22, carreMagique[22]);
SetCtrlVal (pInterface, P_CARRE_C23, carreMagique[23]);
SetCtrlVal (pInterface, P_CARRE_C24, carreMagique[24]);
SetCtrlVal (pInterface, P_CARRE_C25, carreMagique[25]);
SetCtrlVal (pInterface, P_CARRE_C26, carreMagique[26]);
SetCtrlVal (pInterface, P_CARRE_C27, carreMagique[27]);
SetCtrlVal (pInterface, P_CARRE_C28, carreMagique[28]);
SetCtrlVal (pInterface, P_CARRE_C29, carreMagique[29]);
SetCtrlVal (pInterface, P_CARRE_C30, carreMagique[30]);
SetCtrlVal (pInterface, P_CARRE_C31, carreMagique[31]);
SetCtrlVal (pInterface, P_CARRE_C32, carreMagique[32]);
SetCtrlVal (pInterface, P_CARRE_C33, carreMagique[33]);
SetCtrlVal (pInterface, P_CARRE_C34, carreMagique[34]);
SetCtrlVal (pInterface, P_CARRE_C35, carreMagique[35]);
SetCtrlVal (pInterface, P_CARRE_C36, carreMagique[36]);
SetCtrlVal (pInterface, P_CARRE_C37, carreMagique[37]);
SetCtrlVal (pInterface, P_CARRE_C38, carreMagique[38]);
SetCtrlVal (pInterface, P_CARRE_C39, carreMagique[39]);
SetCtrlVal (pInterface, P_CARRE_C40, carreMagique[40]);
SetCtrlVal (pInterface, P_CARRE_C41, carreMagique[41]);
SetCtrlVal (pInterface, P_CARRE_C42, carreMagique[42]);
SetCtrlVal (pInterface, P_CARRE_C43, carreMagique[43]);
SetCtrlVal (pInterface, P_CARRE_C44, carreMagique[44]);
SetCtrlVal (pInterface, P_CARRE_C45, carreMagique[45]);
SetCtrlVal (pInterface, P_CARRE_C46, carreMagique[46]);
SetCtrlVal (pInterface, P_CARRE_C47, carreMagique[47]);
SetCtrlVal (pInterface, P_CARRE_C48, carreMagique[48]);
SetCtrlVal (pInterface, P_CARRE_C49, carreMagique[49]);
SetCtrlVal (pInterface, P_CARRE_C50, carreMagique[50]);
SetCtrlVal (pInterface, P_CARRE_C51, carreMagique[51]);
SetCtrlVal (pInterface, P_CARRE_C52, carreMagique[52]);
SetCtrlVal (pInterface, P_CARRE_C53, carreMagique[53]);
SetCtrlVal (pInterface, P_CARRE_C54, carreMagique[54]);
SetCtrlVal (pInterface, P_CARRE_C55, carreMagique[55]);
SetCtrlVal (pInterface, P_CARRE_C56, carreMagique[56]);
SetCtrlVal (pInterface, P_CARRE_C57, carreMagique[57]);
SetCtrlVal (pInterface, P_CARRE_C58, carreMagique[58]);
SetCtrlVal (pInterface, P_CARRE_C59, carreMagique[59]);
SetCtrlVal (pInterface, P_CARRE_C60, carreMagique[60]);
SetCtrlVal (pInterface, P_CARRE_C61, carreMagique[61]);
SetCtrlVal (pInterface, P_CARRE_C62, carreMagique[62]);
SetCtrlVal (pInterface, P_CARRE_C63, carreMagique[63]);
SetCtrlVal (pInterface, P_CARRE_C64, carreMagique[64]);
SetCtrlVal (pInterface, P_CARRE_C65, carreMagique[65]);
SetCtrlVal (pInterface, P_CARRE_C66, carreMagique[66]);
SetCtrlVal (pInterface, P_CARRE_C67, carreMagique[67]);
SetCtrlVal (pInterface, P_CARRE_C68, carreMagique[68]);
SetCtrlVal (pInterface, P_CARRE_C69, carreMagique[69]);
SetCtrlVal (pInterface, P_CARRE_C70, carreMagique[70]);
SetCtrlVal (pInterface, P_CARRE_C71, carreMagique[71]);
SetCtrlVal (pInterface, P_CARRE_C72, carreMagique[72]);
SetCtrlVal (pInterface, P_CARRE_C73, carreMagique[73]);
SetCtrlVal (pInterface, P_CARRE_C74, carreMagique[74]);
SetCtrlVal (pInterface, P_CARRE_C75, carreMagique[75]);
SetCtrlVal (pInterface, P_CARRE_C76, carreMagique[76]);
SetCtrlVal (pInterface, P_CARRE_C77, carreMagique[77]);
```

```
            SetCtrlVal (pInterface, P_CARRE_C78, carreMagique[78]);
            SetCtrlVal (pInterface, P_CARRE_C79, carreMagique[79]);
            SetCtrlVal (pInterface, P_CARRE_C80, carreMagique[80]);
            SetCtrlAttribute (pInterface, P_CARRE_CB_SAUVER, ATTR_DIMMED, 0);
        }
        else
        {
            SetCtrlAttribute (pInterface, P_CARRE_CB_SAUVER, ATTR_DIMMED, 1);
            MessagePopup ("", "Pas de carré magique trouvé.");
            iRetour = 0;
        }

        return iRetour;
}
//  ____________________________________________________________________________________________________


//  ----------------------------------------------------------------------------------------------------
//  genereCasBase.
//  Fonction permettant la génération aléatoire d'un carré de base respectant les
//      diagonales imposées par l'utilisateur.
//  Paramètres :
//      - pCarreBase    : Carre de base généré.
//      - pOrdre        : Ordre du carré à générer.
//      - pDiag1        : Elements de la diagonale principale descendante.
//      - pDiag2        : Elements de la diagonale principale montante.
//  Valeur retournée : Néant.
//  ----------------------------------------------------------------------------------------------------
void genereCasBase (int pCarreBase [], int pOrdre, int pDiag1[], int pDiag2[])
{
    int i, index1, index2, swap;
    int val1, val2, j, indexVal1, indexVal2;

    //  Initialisation via le carré naturel.
    for (i = 0; i < pOrdre * pOrdre; i++)
        pCarreBase[i] = i + 1;

    //  Mélange aléatoire du carré naturel (Fisher-Yates)
    for (index1 = pOrdre * pOrdre - 1 - 1; index1 > 0; index1--)
    {
        index2 = RoundRealToNearestInteger (rand () / RAND_MAX * (index1 + 1));
        swap = pCarreBase[index1];
        pCarreBase[index1] = pCarreBase[index2];
        pCarreBase[index2] = swap ;
    }
    //  On reconstruit les diagonales principales suivant les spécifications de l'utilisateur.
    //  ------------------------------------------------------------------------------------------------
    //  A chaque itération on traite le ième élément des deux diagonales principales.
    for (i = 0; i < pOrdre; i++)
    {
        indexVal1 = -1;
        indexVal2 = -1;

        val1 = pDiag1[i];
        val2 = pDiag2[i];

        //  On recherche dans le carré les index de l'endroit où se trouve la ième
        //      valeur de la 1ère diagonale
        for (j = 0; j < pOrdre * pOrdre && indexVal1 == -1; j++)
            if (val1 == pCarreBase[j])  indexVal1 = j;

        //  On permute les éléments des index trouvés avec ceux se trouvant sur les
        //      diagonales.
        pCarreBase[indexVal1] = pCarreBase[i * (pOrdre + 1)];
        pCarreBase[i * (pOrdre + 1)] = val1;

        if (val1 != val2)  // Pour éviter de travailler inutilement si on est sur la
                           //  case centrale d'un carré d'ordre impaire.
        {
            //  On recherche dans le carré les index de l'endroit où se trouve la ième
            //      valeur de la 2ème diagonale
            for (j = 0; j < pOrdre * pOrdre && indexVal2 == -1; j++)
                if (val2 == pCarreBase[j])  indexVal2 = j;
```

```
                pCarreBase[indexVal2] = pCarreBase[(i + 1) * (pOrdre - 1)];
                pCarreBase[(i + 1) * (pOrdre - 1)] = val2;
            }
        }
    }
// ____________________________________________________________________________________________


//  ---------------------------------------------------------------------------
//  calculCoutContraintes
//  Fonction permettant le calcul du coût sur les contraintes.
//  Paramètres :
//      - pOrdre : Ordre du carré.
//      - pSomme : Somme magique correspondante.
//      - pCarre : Le carré sur lequel on travaille.
//  Valeur retournée : néant.
//  ---------------------------------------------------------------------------
void ***calculCoutContraintes***(int pOrdre, long pSomme, int pCarre[])
{
    int indexCarre, indexCoutLignes, indexCoutColonnes;
    div_t indexLigne;

    //  Initialisation des coûts sur les contraintes.
    for (indexCarre = 0; indexCarre < pOrdre; indexCarre++)
    {
        coutLignes[indexCarre] = -1 * pSomme;
        coutColonnes[indexCarre] = -1 * pSomme;
    }

    //  Parcours de tous les éléments du carré pour le calcul des contraintes.
    for (indexCarre = 0; indexCarre < pOrdre * pOrdre; indexCarre++)
    {
        //  Mise à jour du coût des contraintes sur la ligne de l'élément traité.
        indexLigne = div (indexCarre, pOrdre);
        indexCoutLignes= indexLigne.quot;
        coutLignes[indexCoutLignes] += pCarre[indexCarre];

        //  Mise à jour du coût des contraintes sur la colonne de l'élément traité.
        indexCoutColonnes = indexCarre % pOrdre;
        coutColonnes[indexCoutColonnes] += pCarre[indexCarre];
    }
}
// ____________________________________________________________________________________________


//  ---------------------------------------------------------------------------------------------
//  MAJCoutContraintes
//  Fonction permettant le calcul de la mise à jourdu coût sur les contraintes
//      après la permutation de deux éléments du carré.
//  Paramètres :
//      - pOrdre : Ordre du carré.
//      - pIndexSwap1 : index de la variable 1.
//      - pValSwap1 : Valeur de la variable 1.
//      - pIndexSwap2 : index de la variable 2.
//      - pValSwap2 : Valeur de la variable 2.
//      - pCarre : Le carré à générer.
//  Attention : les index des variables sont les index APRES permutation
//  Valeur retournée : néant.
//  ---------------------------------------------------------------------------------------------
void ***MAJCoutContraintes*** (int pOrdre, int indexSwap1, int valSwap1, int indexSwap2,
    int valSwap2)
{
    int indexCoutLignes1, indexCoutColonnes1, indexCoutLignes2, indexCoutColonnes2;
    div_t indexLigne;

    //  Récupération des lignes et colonnes sur lesquelles se trouvent les
    //  variables swapée.
    indexLigne = div (indexSwap1, pOrdre);
    indexCoutLignes1 = indexLigne.quot;
    indexLigne = div (indexSwap2, pOrdre);
    indexCoutLignes2 = indexLigne.quot;

    indexCoutColonnes1 = indexSwap1 % pOrdre;
    indexCoutColonnes2 = indexSwap2 % pOrdre;
```

```
    //    Mise à jour des contraintes.
    coutLignes[indexCoutLignes1] =
        coutLignes[indexCoutLignes1] - valSwap2 + valSwap1;
    coutLignes[indexCoutLignes2] =
        coutLignes[indexCoutLignes2] - valSwap1 + valSwap2;

    coutColonnes[indexCoutColonnes1] =
        coutColonnes[indexCoutColonnes1] - valSwap2 + valSwap1;
    coutColonnes[indexCoutColonnes2] =
        coutColonnes[indexCoutColonnes2] - valSwap1 + valSwap2;
}
//  ____________________________________________________________________________________________


//  ------------------------------------------------------------------------------------------------------------
//  calculCoutVariables
//  Fonction permettant le calcul des variables de coût maximum et le coût global
//      de la configuration actuelle du carré.
//  Paramètres :
//      - pCarre : Carre sur lequel on travaille.
//      - pIndexCoutMax : tableau reprenant les index des variables de coût maximum.
//      - pCout : Coût des variables de coût maximum.
//      - pNbCoutsMax : Nombre de variables avec le coût maximum.
//      - pOrdre : Ordre du carré sur lequel on travaille.
//      - pSomme : Somme magique du premier degré pour l'ordre donné.
//      - pCoutGlobal : Coût global pour la configuration actuelle du carré.
//  On utilise les couts sur les contraintes stockés en variable globale.
//  Valeur retournée : Néant.
//  ------------------------------------------------------------------------------------------------------------
void calculCoutVariables (int pCarre[], int pIndexCoutMax[], int *pCout,
    int *pNbCoutsMax, int pOrdre, int pSomme, int *pCoutGlobal)
{
    int indexCarre, indexCoutLignes, indexCoutColonnes;
    div_t indexLigne;
    int coutLocal, coutLocalMax;

    coutLocalMax = -1;

    //    Parcours du carré pour le calcul du coût des variables.
    for (indexCarre = 0; indexCarre < pOrdre * pOrdre; indexCarre++)
    {
        //    On vérifie que l'élément traité n'est pas tabu.
        if (listeTabu[indexCarre] == 0)
        {
            //    Mise à jour du coût de l'élément traité en fonction de la ligne
            //        où il se trouve.
            indexLigne = div (indexCarre, pOrdre);
            indexCoutLignes= indexLigne.quot;
            coutLocal = coutLignes[indexCoutLignes];

            //    Mise à jour du coût de l'élément traité en fonction de la colonne
            //        où il se trouve.
            indexCoutColonnes = indexCarre % pOrdre;
            coutLocal += coutColonnes[indexCoutColonnes];

            //    On utilise la valeur absolue du coût calculé
            if (coutLocal < 0) coutLocal *= -1;

            //    L'élément traité a un coup équivalent au coût maximum actuel du carré.
            //    --> On l'ajoute au vecteur des éléments de coût maximum.
            if (coutLocal == coutLocalMax)
            {
                pIndexCoutMax[*pNbCoutsMax] = indexCarre;
                (*pNbCoutsMax)++;
            }
```

```
                //    L'élément traité à un coût supérieur au coût maximu actuel du carré.
                //    -->  On ré-initialise le vecteur des variable de coût max avec l'élément.
                if (coutLocal > coutLocalMax)
                {
                    coutLocalMax = coutLocal;
                    pIndexCoutMax[0] = indexCarre;
                    *pNbCoutsMax = 1;
                }
            }
        }

        //    Calcul du coût global de la configuration actuelle du carré.
        //    (par addition de la valeur absolue de tous les coûts sur les contraintes)
        *pCoutGlobal = 0;
        for (indexCarre = 0; indexCarre < pOrdre; indexCarre++)
        {
            if (coutLignes[indexCarre] < 0)
                *pCoutGlobal += -1 * coutLignes[indexCarre];
            else
                *pCoutGlobal += coutLignes[indexCarre];

            if (coutColonnes[indexCarre] < 0)
                *pCoutGlobal += -1 * coutColonnes[indexCarre];
            else
                *pCoutGlobal += coutColonnes[indexCarre];
        }

        //    On garni les variables de retour vers la fonction appelante.
        *pCout = coutLocalMax;
}
//    ____________________________________________________________________________________________


//    ------------------------------------------------------------------------------------------------------------------
//    fCoutMinimum
//    Fonction permettant de calculer la configuration voisine de coût minimum.  Si plusieurs
//        configurations sont à égalité, l'une d'entre-elle est choisie au hasard.
//    Paramètres :
//        - pCarre                    : Le carré en cours de travail.
//        - pOrdre                    : L'ordre du carré en cours de travail.
//        - indexCoutMin (réf.)       : Element dont la permutation avec l'élément choisi permet
//            d'obtenir la configuration voisine de coût minimum.
//        - nouveauCout (réf.)        : Coût global de la configuration de coût minimum trouvée.
//        - indexVariableAPermuter: Index de la variable servant à la recherche de la
//            configuration voisine de coût minimum.
//        - pSomme                    : Constante magique de l'ordre du carré.
//        - pCoutGlobal               : Coût global de la configuration courante.
//    Valeur retournée : Néant
//    ------------------------------------------------------------------------------------------------------------------
void fCoutMinimum (int pCarre[], int pOrdre, int *indexCoutMin, int *nouveauCout,
    int indexVariableAPermuter, int pSomme, int pCoutGlobal)
{
    int valSwap;
    int indexCourant;
    int coutConfig, coutMin;
    int indexCoutMinLocal[ORDRE_CARRE * ORDRE_CARRE];    //    Stockage des index de coûts min temporaires.
    int nbCoutsMin; //    Nombre de fois où la config de coût min. est trouvée.
    double randomMax, resultatRandom;

    //    Parcours de tous les swaps possibles.
    // On initialise le cout minimum au cout global courant - 1 pour éviter de boucler.
    coutMin = pCoutGlobal - 1;
    nbCoutsMin = 0;
    valSwap = pCarre[indexVariableAPermuter];
    for (indexCourant = 0; indexCourant < pOrdre * pOrdre; indexCourant++)
    {
        //    Pas de permutation sur le pivot ni sur les variables Tabu.
        if (indexCourant == indexVariableAPermuter || listeTabu[indexCourant] != 0)
            continue;
```

```
        //    Pas de permutation si l'élément à permuter est sur la diagonale descendante
        //          et pas l'élément dont la configuration est testée ou vice-versa.
        if ((!(indexVariableAPermuter % (pOrdre + 1)) && (indexCourant % (pOrdre + 1)))
            || ((indexVariableAPermuter % (pOrdre + 1)) && !(indexCourant % (pOrdre + 1))))
            continue;

        //    Pas de permutation si l'élément à permuter est sur la diagonale montante
        //          et pas l'élément dont la configuration est testée ou vice-versa.
        if ((!(indexVariableAPermuter % (pOrdre - 1)) && (indexCourant % (pOrdre - 1)))
            || ((indexVariableAPermuter % (pOrdre - 1)) && !(indexCourant % (pOrdre - 1))))
            continue;

        //    Dans les autres cas, on teste la permutation.

        //    Permutation dans le carre.
        pCarre[indexVariableAPermuter] = pCarre[indexCourant];
        pCarre[indexCourant] = valSwap;
        MAJCoutContraintes(pOrdre, indexVariableAPermuter,
            pCarre[indexVariableAPermuter], indexCourant,pCarre[indexCourant]);

        //    Calcul du coût global sur le carré.
        fCoutGlobal (pCarre, pOrdre, pSomme, &coutConfig);

        //    Nouveau cout équivalent au cout min trouvé.
        if (coutConfig == coutMin)
        {
            indexCoutMinLocal[nbCoutsMin] = indexCourant;
            nbCoutsMin++;
        }

        //    Nouveau coût minimum trouvé.
        if (coutConfig < coutMin)
        {
            coutMin = coutConfig;
            nbCoutsMin = 1;
            indexCoutMinLocal[0] = indexCourant;
        }

        //    Retour au carré d'origine.
        pCarre[indexCourant] = pCarre[indexVariableAPermuter];
        pCarre[indexVariableAPermuter] = valSwap;
        MAJCoutContraintes(pOrdre, indexVariableAPermuter,
            pCarre[indexVariableAPermuter], indexCourant,pCarre[indexCourant]);
    }

    //    Choix aléatoire d'une configuration voisine de coût minimum trouvée.
    if (nbCoutsMin > 0)
    {
        randomMax = nbCoutsMin - 1;
        resultatRandom = rand ();
        resultatRandom /= RAND_MAX;
        resultatRandom *= randomMax;
        indexCourant = RoundRealToNearestInteger (resultatRandom);
        *indexCoutMin = indexCoutMinLocal[indexCourant];
    }
    else *indexCoutMin = -1;  // Cas où pas de nouveaux cout min trouvé
    *nouveauCout = coutMin;
}
//    ____________________________________________________________________________
```

```
//  ---------------------------------------------------------------------------------------------------
//  fCoutGlobal
//  Fonction permettant le calcul du coût global de la configuration actuelle.
//  Paramètres :
//      - pCarre        : Carré en cour de travail.
//      - pOrdre        : Ordre de grandeur du carré.
//      - pSomme        : Constante magique de l'ordre du carré.
//      - pCout (réf.)  : Cout global calculé.
//  La variable utilise les coûts sur les contraintes stockées en global.
//  Valeur retournée : Néant
//  ---------------------------------------------------------------------------------------------------
void fCoutGlobal (int pCarre[], int pOrdre, int pSomme, int *pCout)
{
    int indexCarre, indexCoutLignes;
    int indexCoutColonnes;
    div_t indexLigne;

    //  Calcul du cout global.
    *pCout = 0;
    for (indexCarre = 0; indexCarre < pOrdre; indexCarre++)
    {
        if (coutLignes[indexCarre] < 0)
            *pCout += -1 * coutLignes[indexCarre];
        else
            *pCout += coutLignes[indexCarre];

        if (coutColonnes[indexCarre] < 0)
            *pCout += -1 * coutColonnes[indexCarre];
        else
            *pCout += coutColonnes[indexCarre];
    }
}
//  _____

//  ---------------------------------------------------------------------------------------------------
//  fResetPartiel
//  Fonction permettant de ré-initialiser une partie aléatoire des éléments du carré.
//  Paramètres :
//      - pCarre : Vecteur contenant les éléments du carré sur lequel on travaille.
//      - pOrdre : Ordre de grandeur du carré.
//      - pNbVariables : Nombre de variables à ré-initialiser.
//  Valeur retournée : Néant
//  ---------------------------------------------------------------------------------------------------
void fResetPartiel (int pCarre[], int pOrdre, int pNbVariables)
{
    int listeReset[NB_VARIABLES_RESET];     //  Vecteur dans lequel se trouve les indices des variables à reseter
    long i;
    double randomMax, resultatRandom;
    int indiceAPermuter1, indiceAPermuter2, isNoyau, swap;
    long nbPermutations;

    //  Génération aléatoire des index des variables à mélanger.
    for (i = 0; i < pNbVariables; i++)
    {
        randomMax = pOrdre * pOrdre - 1;
        resultatRandom = rand ();
        resultatRandom /= RAND_MAX;
        resultatRandom *= randomMax;
        resultatRandom = RoundRealToNearestInteger (resultatRandom);
        //  On vérifie que ce n'est pas un élément bloqué.
        if (!(pOrdre % 2))
        {
            if ((resultatRandom == (pOrdre / 2 - 1) * (pOrdre + 1)) ||
                (resultatRandom == pOrdre / 2 * (pOrdre - 1)) ||
                (resultatRandom == (pOrdre / 2 + 1) * (pOrdre - 1)) ||
                (resultatRandom == pOrdre / 2 * (pOrdre + 1)))
                //  C'est un élément du noyau.
                i--;
            else
                listeReset[i] = RoundRealToNearestInteger (resultatRandom);
        }
        else
```

```
        {
            if (resultatRandom == (pOrdre * pOrdre - 1) / 2)
                //    C'est l'élément central.
                i--;
            else
                listeReset[i] = RoundRealToNearestInteger (resultatRandom);
        }
    }

    //    Permutations aléatoires des éléments d'indices correspondants.
    nbPermutations = pow (pOrdre, 2);
    for (i = 0; i < nbPermutations; i++)
    {
        //    Choix du premier index au hasard.
        randomMax = pNbVariables - 1;
        resultatRandom = rand ();
        resultatRandom /= RAND_MAX;
        resultatRandom *= randomMax;
        indiceAPermuter1 = RoundRealToNearestInteger (resultatRandom);

        //    Si l'index choisi se trouve sur la diagonale montante.
        if (listeReset[indiceAPermuter1] % (pOrdre - 1) == 0 &&
                listeReset[indiceAPermuter1] != 0 &&
                listeReset[indiceAPermuter1] != pOrdre * pOrdre - 1)
        {
            do
            {
                // Génération au hasard d'un autre indice de la diagonale.
                randomMax = pOrdre - 1;
                resultatRandom = rand ();
                resultatRandom /= RAND_MAX;
                resultatRandom *= randomMax;
                indiceAPermuter2 = RoundRealToNearestInteger (resultatRandom);

                //    Vérification que ce n'est pas un élément bloqué.
                isNoyau = 0;
                if (!(pOrdre % 2))
                {
                    if (((indiceAPermuter2 + 1) * (pOrdre - 1) == (pOrdre / 2 - 1) * (pOrdre + 1)) ||
                        ((indiceAPermuter2 + 1) * (pOrdre - 1) == pOrdre / 2 * (pOrdre - 1)) ||
                        ((indiceAPermuter2 + 1) * (pOrdre - 1) == (pOrdre / 2 + 1) * (pOrdre - 1)) ||
                        ((indiceAPermuter2 + 1) * (pOrdre - 1) == pOrdre / 2 * (pOrdre + 1)))
                            //    C'est un élément du noyau.
                            isNoyau = 1;
                }
                else
                {
                    if ((indiceAPermuter2 + 1) * (pOrdre - 1) == (pOrdre * pOrdre - 1) / 2)
                        //    C'est l'élément central.
                        isNoyau = 1;
                }
            }
            while (isNoyau == 1);

            //    Permutation.
            swap                                        = pCarre[listeReset[indiceAPermuter1]];
            pCarre[listeReset[indiceAPermuter1]]        = pCarre[(indiceAPermuter2 + 1) * (pOrdre - 1)];
            pCarre[(indiceAPermuter2 + 1) * (pOrdre - 1)]    = swap;

            //    Mise à jour des contraintes.
            MAJCoutContraintes(pOrdre, listeReset[indiceAPermuter1],
                pCarre[listeReset[indiceAPermuter1]],
                (indiceAPermuter2 + 1) * (pOrdre - 1),
                pCarre[(indiceAPermuter2 + 1) * (pOrdre - 1)]);

            //    Mise à jour éventuelle de la liste Tabou.
            if (listeTabu[listeReset[indiceAPermuter1]] > 0)
            {
                listeTabu[listeReset[indiceAPermuter1]] = 0;
                nbVariablesTabou--;
            }
```

```
        if (listeTabu[(indiceAPermuter2 + 1) * (pOrdre - 1)] > 0)
        {
            listeTabu[(indiceAPermuter2 + 1) * (pOrdre - 1)] = 0;
            nbVariablesTabou--;
        }
    }

    //    Si l'index choisi se trouve sur la diagonale descendante.
    if (!(listeReset[indiceAPermuter1] % (pOrdre + 1)))
    {
        do
        {
            // Génération au hasard d'un autre indice de la diagonale.
            randomMax = pOrdre - 1;
            resultatRandom = rand ();
            resultatRandom /= RAND_MAX;
            resultatRandom *= randomMax;
            indiceAPermuter2 = RoundRealToNearestInteger (resultatRandom);
            //    Vérification que ce n'est pas un élément bloqué.
            isNoyau = 0;
            if (!(pOrdre % 2))
            {
                if ((indiceAPermuter2 * (pOrdre + 1) == (pOrdre / 2 - 1) * (pOrdre + 1)) ||
                    (indiceAPermuter2 * (pOrdre + 1) == pOrdre / 2 * (pOrdre - 1)) ||
                    (indiceAPermuter2 * (pOrdre + 1) == (pOrdre / 2 + 1) * (pOrdre - 1)) ||
                    (indiceAPermuter2 * (pOrdre + 1) == pOrdre / 2 * (pOrdre + 1)))
                        //    C'est un élément du noyau.
                        isNoyau = 1;
            }
            else
            {
                if (indiceAPermuter2 * (pOrdre + 1) == (pOrdre * pOrdre - 1) / 2)
                    //    C'est l'élément central.
                    isNoyau = 1;
            }
        }
        while (isNoyau == 1);

        //    Permutation.
        swap                                        = pCarre[listeReset[indiceAPermuter1]];
        pCarre[listeReset[indiceAPermuter1]]    = pCarre[indiceAPermuter2 * (pOrdre + 1)];
        pCarre[indiceAPermuter2 * (pOrdre + 1)]     = swap;

        //    Mise à jour des contraintes.
        MAJCoutContraintes(pOrdre, listeReset[indiceAPermuter1],
            pCarre[listeReset[indiceAPermuter1]],
            indiceAPermuter2 * (pOrdre + 1),
            pCarre[indiceAPermuter2 * (pOrdre + 1)]);

        //    Mise à jour éventuelle de la liste Tabou.
        if (listeTabu[listeReset[indiceAPermuter1]] > 0)
        {
            listeTabu[listeReset[indiceAPermuter1]] = 0;
            nbVariablesTabou--;
        }
        if (listeTabu[indiceAPermuter2 * (pOrdre + 1)] > 0)
        {
            listeTabu[indiceAPermuter2 * (pOrdre + 1)] = 0;
            nbVariablesTabou--;
        }
    }
    //    S'il se trouve à un autre endroit du carré.
    if ((listeReset[indiceAPermuter1] % (pOrdre - 1)) && (listeReset[indiceAPermuter1] % (pOrdre + 1)))
    {
        do
        {
            //    Choix d'un deuxième indice au hasard (hors diagonales.)
            randomMax = pNbVariables - 1;
            resultatRandom = rand ();
            resultatRandom /= RAND_MAX;
            resultatRandom *= randomMax;
            indiceAPermuter2 = RoundRealToNearestInteger (resultatRandom);
        }
        while  (!(listeReset[indiceAPermuter2] % (pOrdre - 1)) || !(listeReset[indiceAPermuter2] % (pOrdre + 1)));
```

```
            //    Permutation.
            swap                                    = pCarre[listeReset[indiceAPermuter1]];
            pCarre[listeReset[indiceAPermuter1]]    = pCarre[listeReset[indiceAPermuter2]];
            pCarre[listeReset[indiceAPermuter2]]    = swap;

            //    Mise à jour des contraintes.
            MAJCoutContraintes(pOrdre, listeReset[indiceAPermuter1],
                pCarre[listeReset[indiceAPermuter1]],
                listeReset[indiceAPermuter2],
                pCarre[listeReset[indiceAPermuter2]]);

            //    Mise à jour éventuelle de la liste Tabou.
            if (listeTabu[listeReset[indiceAPermuter1]] > 0)
            {
                listeTabu[listeReset[indiceAPermuter1]] = 0;
                nbVariablesTabou--;
            }
            if (listeTabu[listeReset[indiceAPermuter2]] > 0)
            {
                listeTabu[listeReset[indiceAPermuter2]] = 0;
                nbVariablesTabou--;
            }
        }
    }
}
//    ____________________________________________________________________________________________


//    ---------------------------------------------------------------------------
//    Gestion de la liste Tabu
//    ---------------------------------------------------------------------------

//    ---------------------------------------------------------------------------------------------------------------------
//    fInitTabu
//    Initialisation de la liste Tabu.
//    Paramètres : Néant.
//    Valeur retournée : Néant.
//    Environnement : On utilise un tableau global de taille ORDRE_CARRE * ORDRE_CARRE
//          comme liste Tabu.
//    ---------------------------------------------------------------------------------------------------------------------
void fInitTabu (void)
{
    int i;

    //    On initialise tous les éléments du carré à Non-tabu.
    for (i = 0; i < ORDRE_CARRE * ORDRE_CARRE; i++)
        listeTabu[i] = 0;

    //    On bloque le noyau en le déclarant Tabu pour la durée de l'algorithme.
    if (!(ORDRE_CARRE % 2))
    {
        //    C'est un ordre pair --> On fixe les 4 éléments centraux du carré.
        //    On fixe le carré dans l'ordre :
        //              X   X   X   X
        //              X   1   2   X
        //              X   3   4   X
        //              X   X   X   X
        listeTabu[(ORDRE_CARRE / 2 - 1) * (ORDRE_CARRE + 1)]     = NB_ITERATIONS + 1;
        listeTabu[ORDRE_CARRE / 2 * (ORDRE_CARRE - 1)]           = NB_ITERATIONS + 1;
        listeTabu[(ORDRE_CARRE / 2 + 1) * (ORDRE_CARRE - 1)]     = NB_ITERATIONS + 1;
        listeTabu[ORDRE_CARRE / 2 * (ORDRE_CARRE + 1)]               = NB_ITERATIONS + 1;
        nbVariablesTabou = 4;  //    Initialisation du nombre de variables Tabou
    }
    else
    {
        //    C'est un ordre impair. -->  On fixe l'élément central
        listeTabu[(ORDRE_CARRE * ORDRE_CARRE - 1) / 2] = NB_ITERATIONS + 1;
        nbVariablesTabou = 1;  //    Initialisation du nombre de variables Tabou
    }
}
//    ____________________________________________________________________________________________
```

```
//  ---------------------------------------------------------------------------------------------------------------------
//  fAddVariable
//  Fonction permettant d'ajouter une variable à la liste Tabu.
//  Paramètres :
//      - pIndex : Index de la variable à mettre dans la liste Tabu.
//  Valeur retournée : Néant.
//  Environnement :
//      - On utilise un tableau global de taille ORDRE_CARRE * ORDRE_CARRE  comme
//          liste Tabu.
//      - On utilise la constante NB_ITER_EXCL pour définir le nombre d'itérations
//          d'exclusion (= nombre d'itérations où la variable est dans la liste Tabu)
//  ---------------------------------------------------------------------------------------------------------------------
void fAddVariable (int pIndex)
{
    listeTabu[pIndex] = NB_ITER_EXCL;
    //  On incrémente le nombre de variables Tabou.
    nbVariablesTabou++;
}
//  _____________________________________________________________________________


//  ---------------------------------------------------------------------------------------------------------------------
//  fMAJTabu
//  Fonction de mise à jour de la liste tabu lors de chaque itération.
//  Paramètres : Néant.
//  Valeur retournée : Néant.
//  Environnement : On utilise un tableau global de taille ORDRE_CARRE * ORDRE_CARRE
//      comme liste Tabu
//  ---------------------------------------------------------------------------------------------------------------------
void fMAJTabu (void)
{
    int i;

    //  Ré-initialisation du nombre de variables tabou.
    nbVariablesTabou = 0;

    //  Mise à jour = décrémentation de 1.  On ne passe jamais en dessous de 0.
    //  On profite du parcours du vecteur pour mettre à jour le nombre de variables Tabou.
    for (i = 0; i < ORDRE_CARRE * ORDRE_CARRE; i++)
    {
        listeTabu[i] = Max (0, listeTabu[i] - 1);
        if (listeTabu[i] > 0) nbVariablesTabou++;
    }
}
//  _____________________________________________________________________________


//  ---------------------------------------------------------------------------------------------------------------------
//  fVerifValiditeDiag
//  Fonction permettant de vérifier que les diagonales encodées sont correctes.
//  Paramètres :
//      - pDiag1 : Tableau contenant les éléments de la 1ère diagonale.
//      - pDiag2 : Tableau contenant les éléments de la 2ème diagonale.
//      - pOrdre : Ordre du carré.
//  Valeur retournée : la fonction retourne 1 si les diagonales sont correctes
//      ou 0 si elles ne le sont pas.
//  ---------------------------------------------------------------------------------------------------------------------
int fVerifValiditeDiag (int pDiag1[], int pDiag2[], int pOrdre)
{
    int i, j, iErreur;
    long sommeD1, sommeD2, sommeMagique;

    iErreur = 0;
    sommeD1 = sommeD2 = 0;

    //  Parcours des diagonales.
    for (i = 0; i < pOrdre && iErreur == 0; i++)
    {
        //  Vérification de l'unicité des éléments.
        for (j = i + 1; j < pOrdre; j++)
        {
            if (pDiag1[i] == pDiag1[j])    iErreur = 1;
            if (pDiag2[i] == pDiag2[j])    iErreur = 1;
        }
```

```
            //    Calcul de la somme sur chacune des diagonales.
            sommeD1 += pDiag1[i];
            sommeD2 += pDiag2[i];
        }

        //    FVérification que les diagonales sont bien magiques.
        if (iErreur == 0)
        {
            sommeMagique = pOrdre * (pOrdre * pOrdre + 1) / 2;
            if (sommeD1 != sommeMagique || sommeD2 != sommeMagique)
                iErreur = 1;
        }
        return (!iErreur);
}
//    ____________________________________________________________________________


//    ---------------------------------------------------------------------------------------------------------------------
//    fSauveCarre
//    Permet de sauvegarder le carré généré dans un fichier séparé par des ';'
//    Paramètres :
//        - pInterface : Interface sur lequel le carré généré est affiché.
//        - pOrdre : Ordre du carré magique.
//    Valeur retournée : La fonction retourne 0 en cas de succès ou -1 en cas d'échec de la
//        sauvegarde.
//    ---------------------------------------------------------------------------------------------------------------------
int fSauveCarre (int pInterface, int pOrdre)
{
        int i, j, fileHandle, erreur;
        char afficheCarre[500];
        int carreMagique[ORDRE_CARRE * ORDRE_CARRE];

        //    Récupération des données du carré dans un vecteur.
        GetCtrlVal (pInterface, P_CARRE_C0, &carreMagique[0]);
        GetCtrlVal (pInterface, P_CARRE_C1, &carreMagique[1]);
        GetCtrlVal (pInterface, P_CARRE_C2, &carreMagique[2]);
        GetCtrlVal (pInterface, P_CARRE_C3, &carreMagique[3]);
        GetCtrlVal (pInterface, P_CARRE_C4, &carreMagique[4]);
        GetCtrlVal (pInterface, P_CARRE_C5, &carreMagique[5]);
        GetCtrlVal (pInterface, P_CARRE_C6, &carreMagique[6]);
        GetCtrlVal (pInterface, P_CARRE_C7, &carreMagique[7]);
        GetCtrlVal (pInterface, P_CARRE_C8, &carreMagique[8]);
        GetCtrlVal (pInterface, P_CARRE_C9, &carreMagique[9]);
        GetCtrlVal (pInterface, P_CARRE_C10, &carreMagique[10]);
        GetCtrlVal (pInterface, P_CARRE_C11, &carreMagique[11]);
        GetCtrlVal (pInterface, P_CARRE_C12, &carreMagique[12]);
        GetCtrlVal (pInterface, P_CARRE_C13, &carreMagique[13]);
        GetCtrlVal (pInterface, P_CARRE_C14, &carreMagique[14]);
        GetCtrlVal (pInterface, P_CARRE_C15, &carreMagique[15]);
        GetCtrlVal (pInterface, P_CARRE_C16, &carreMagique[16]);
        GetCtrlVal (pInterface, P_CARRE_C17, &carreMagique[17]);
        GetCtrlVal (pInterface, P_CARRE_C18, &carreMagique[18]);
        GetCtrlVal (pInterface, P_CARRE_C19, &carreMagique[19]);
        GetCtrlVal (pInterface, P_CARRE_C20, &carreMagique[20]);
        GetCtrlVal (pInterface, P_CARRE_C21, &carreMagique[21]);
        GetCtrlVal (pInterface, P_CARRE_C22, &carreMagique[22]);
        GetCtrlVal (pInterface, P_CARRE_C23, &carreMagique[23]);
        GetCtrlVal (pInterface, P_CARRE_C24, &carreMagique[24]);
        GetCtrlVal (pInterface, P_CARRE_C25, &carreMagique[25]);
        GetCtrlVal (pInterface, P_CARRE_C26, &carreMagique[26]);
        GetCtrlVal (pInterface, P_CARRE_C27, &carreMagique[27]);
        GetCtrlVal (pInterface, P_CARRE_C28, &carreMagique[28]);
        GetCtrlVal (pInterface, P_CARRE_C29, &carreMagique[29]);
        GetCtrlVal (pInterface, P_CARRE_C30, &carreMagique[30]);
        GetCtrlVal (pInterface, P_CARRE_C31, &carreMagique[31]);
        GetCtrlVal (pInterface, P_CARRE_C32, &carreMagique[32]);
        GetCtrlVal (pInterface, P_CARRE_C33, &carreMagique[33]);
        GetCtrlVal (pInterface, P_CARRE_C34, &carreMagique[34]);
        GetCtrlVal (pInterface, P_CARRE_C35, &carreMagique[35]);
        GetCtrlVal (pInterface, P_CARRE_C36, &carreMagique[36]);
        GetCtrlVal (pInterface, P_CARRE_C37, &carreMagique[37]);
        GetCtrlVal (pInterface, P_CARRE_C38, &carreMagique[38]);
        GetCtrlVal (pInterface, P_CARRE_C39, &carreMagique[39]);
```

```
    GetCtrlVal (pInterface, P_CARRE_C40, &carreMagique[40]);
    GetCtrlVal (pInterface, P_CARRE_C41, &carreMagique[41]);
    GetCtrlVal (pInterface, P_CARRE_C42, &carreMagique[42]);
    GetCtrlVal (pInterface, P_CARRE_C43, &carreMagique[43]);
    GetCtrlVal (pInterface, P_CARRE_C44, &carreMagique[44]);
    GetCtrlVal (pInterface, P_CARRE_C45, &carreMagique[45]);
    GetCtrlVal (pInterface, P_CARRE_C46, &carreMagique[46]);
    GetCtrlVal (pInterface, P_CARRE_C47, &carreMagique[47]);
    GetCtrlVal (pInterface, P_CARRE_C48, &carreMagique[48]);
    GetCtrlVal (pInterface, P_CARRE_C49, &carreMagique[49]);
    GetCtrlVal (pInterface, P_CARRE_C50, &carreMagique[50]);
    GetCtrlVal (pInterface, P_CARRE_C51, &carreMagique[51]);
    GetCtrlVal (pInterface, P_CARRE_C52, &carreMagique[52]);
    GetCtrlVal (pInterface, P_CARRE_C53, &carreMagique[53]);
    GetCtrlVal (pInterface, P_CARRE_C54, &carreMagique[54]);
    GetCtrlVal (pInterface, P_CARRE_C55, &carreMagique[55]);
    GetCtrlVal (pInterface, P_CARRE_C56, &carreMagique[56]);
    GetCtrlVal (pInterface, P_CARRE_C57, &carreMagique[57]);
    GetCtrlVal (pInterface, P_CARRE_C58, &carreMagique[58]);
    GetCtrlVal (pInterface, P_CARRE_C59, &carreMagique[59]);
    GetCtrlVal (pInterface, P_CARRE_C60, &carreMagique[60]);
    GetCtrlVal (pInterface, P_CARRE_C61, &carreMagique[61]);
    GetCtrlVal (pInterface, P_CARRE_C62, &carreMagique[62]);
    GetCtrlVal (pInterface, P_CARRE_C63, &carreMagique[63]);
    GetCtrlVal (pInterface, P_CARRE_C64, &carreMagique[64]);
    GetCtrlVal (pInterface, P_CARRE_C65, &carreMagique[65]);
    GetCtrlVal (pInterface, P_CARRE_C66, &carreMagique[66]);
    GetCtrlVal (pInterface, P_CARRE_C67, &carreMagique[67]);
    GetCtrlVal (pInterface, P_CARRE_C68, &carreMagique[68]);
    GetCtrlVal (pInterface, P_CARRE_C69, &carreMagique[69]);
    GetCtrlVal (pInterface, P_CARRE_C70, &carreMagique[70]);
    GetCtrlVal (pInterface, P_CARRE_C71, &carreMagique[71]);
    GetCtrlVal (pInterface, P_CARRE_C72, &carreMagique[72]);
    GetCtrlVal (pInterface, P_CARRE_C73, &carreMagique[73]);
    GetCtrlVal (pInterface, P_CARRE_C74, &carreMagique[74]);
    GetCtrlVal (pInterface, P_CARRE_C75, &carreMagique[75]);
    GetCtrlVal (pInterface, P_CARRE_C76, &carreMagique[76]);
    GetCtrlVal (pInterface, P_CARRE_C77, &carreMagique[77]);
    GetCtrlVal (pInterface, P_CARRE_C78, &carreMagique[78]);
    GetCtrlVal (pInterface, P_CARRE_C79, &carreMagique[79]);
    GetCtrlVal (pInterface, P_CARRE_C80, &carreMagique[80]);

    //    Création de la chaîne de caractères à sauver.
    Fmt (afficheCarre, "%s<%s", "");
    for (i = 0; i < pOrdre; i++)
        for (j = 0; j < pOrdre; j++)
            if (j == pOrdre - 1)
                Fmt (afficheCarre, "%s<%s%d[w2p0]\n", afficheCarre,
                        carreMagique[i * pOrdre + j]);
            else
                Fmt (afficheCarre, "%s<%s%d[w2p0];", afficheCarre,
                        carreMagique[i * pOrdre + j]);

    //    Sauvegarde dans le fichier.
    fileHandle = OpenFile ("data9.txt", VAL_WRITE_ONLY, VAL_TRUNCATE, VAL_ASCII);
    if (fileHandle == -1) return  -1;
    erreur = WriteFile (fileHandle, afficheCarre, StringLength (afficheCarre));
    if (erreur == -1) return -1;
    CloseFile (fileHandle);

    return 0;
}
//  ____________________________________________________________________________
```

A.2 Code source du programme de génération de lignes magiques

```
//  ---------------------------------------------------------------------------------------------------------
//  Programme permettant la génération de l'ensemble des lignes magiques d'un ordre
//  donné ainsi que la vérification des deuxième et troisième ordres de magie.
//  Auteur : Leblanc J-N
//  ---------------------------------------------------------------------------------------------------------

//  Inclusion des fichiers d'en-têtes nécessaires.
//  ----------------------------------------------------------------
#include <formatio.h>
#include <userint.h>
#include <ansi_c.h>

//  Prototype des fonctions de la librairie.
//  ----------------------------------------------------
void fCalculIteratif (int nbTerme, int somme, int xMax, int nbElemCalcule, int pOrdre);
int isDeuxiemeOrdre (int pVecteur[], int pOrdre);
int isTroisiemeOrdre (int pVecteur[], int pOrdre);

//  Déclaration des variables globales.
long nbLignes;
int tableauResultats[15];

//  Fonction principale.
//  ---------------------------
int main (int argc, char * argv[])
{
    int ordre, somme, x1min, x1max, x1;
    long si;
    void *entreeListe = NULL, *listeCourante = NULL;

    //  Vérification des arguments.
    if (argc != 2)
    {
        MessagePopup ("Argument incorrect", "Vous devez passer l'ordre en paramètre.");
        return -1;
    }
    //  Récupération de l'ordre passé en paramètre.
    ordre = atoi (argv[1]);

    //  Calcul de la somme magique.
    somme = ordre * (ordre * ordre + 1) / 2;

    if (ordre >= 3)
    {
        //  Calcul des valeurs max et min pour l'élément le plus grand de la ligne.
        x1max = ordre * ordre;
        si = (ordre - 1) * ordre  / 2;
        x1min = (int)((somme + si) / ordre);
        if ((somme + si) % ordre) x1min++;

        //  Lancement des itérations récursive pour chaque valeur de x1.
        nbLignes = 1;
        for (x1 = x1max; x1 >= x1min; x1--)
        {
            //  Traitement.
            tableauResultats[1] = x1;

            //  Calcul itératif avec le nombre fixé.
            fCalculIteratif (ordre - 1, somme - x1, x1, 1, ordre);
        }
    }
    nbLignes--; // Il a toujours compté une ligne de trop.
    return 0;
}
//  ___________________________________________________________________________________________
```

```
//  ---------------------------------------------------------------------------------------------------------------
//  fCalculIteratif
//  Fonction permettant de calculer récursivement la partie restante d'une ligne magique.
//  Paramètres :
//      - nbTerme : Nombre de termes encore à calculer sur la ligne magique en cours.
//      - somme : Somme que doivent faire les terme restant à calculer.
//      - xMax : Valeur maximum que peuvent prendre les termes restant à calculer.
//      - nbElemCalcule : Nombre d'éléments de la ligne déjà connus.
//      - pOrdre : Ordre des lignes magiques recherchées.
//  ---------------------------------------------------------------------------------------------------------------
void fCalculIteratif (int nbTerme, int somme, int xMax, int nbElemCalcule, int pOrdre)
{
    int xiMin, xiMax, xi, handle;
    long si;

    if (nbTerme > 1)
    {
        //  Calcul de la valeur minimum de ce terme
        si = (nbTerme -1) * nbTerme / 2;
        xiMin = (int)((somme + si) / nbTerme);
        if ((somme + si) % nbTerme) xiMin++;

        //  Calcul de la valeur maximum de ce terme.
        if (somme < xMax)
            xiMax = somme - 1;
        else
            xiMax = xMax - 1;

        //  Parcours des différentes valeurs possible pour ce terme.
        for (xi = xiMax; xi >= xiMin; xi--)
        {
            //  Ajout de la valeur calculée dans le vecteur résultat.
            tableauResultats[nbElemCalcule + 1] = xi;

            //  Appel récurent pour le reste du vecteur.
            fCalculIteratif (nbTerme - 1, somme - xi, xi, nbElemCalcule + 1, pOrdre);
        }
    }
    else    // Dernier terme du vecteur.
    {
        //  Ajout de la valeur calculée dans le vecteur résultat.
        tableauResultats[nbElemCalcule + 1] = somme;

        //  Ajout des caractéristiques supplémentaires.
        tableauResultats[0] = nbLignes;
        tableauResultats[pOrdre + 1] = isDeuxiemeOrdre (tableauResultats, pOrdre);
        tableauResultats[pOrdre + 2] = isTroisiemeOrdre (tableauResultats, pOrdre);
        //  Mise à jour du fichier de données.
        ArrayToFile ("data.txt", tableauResultats, VAL_INTEGER, pOrdre + 3, 1,
            VAL_GROUPS_TOGETHER, VAL_GROUPS_AS_ROWS, VAL_SEP_BY_TAB, 10,
            VAL_ASCII, VAL_APPEND);

        //  Passage à la ligne suivante
        nbLignes++;
    }
}
//  ___________________________________________________________________________________________________________
```

```
//  ---------------------------------------------------------------------------------------------------
//  isDeuxiemeOrdre
//  Fonction permettant de définir si une ligne magique est bimagique
//  Paramètres :
//      - pVecteur : Ligne magique à tester
//      - pOrdre : ordre de grandeur de la ligne à vérifier.
//  Valeur retournée : La fonction retourne 1 si la ligne est bimagique ou 0 si elle
//      est simplement magique.
//  ---------------------------------------------------------------------------------------------------
int isDeuxiemeOrdre (int pVecteur[], int pOrdre)
{
    //  Taille du vecteur : minimum ordre + 1
    //  Element du vecteur dont on tient compte : 1 -> ordre
    int k2, sommeVecteur, i, retour;

    //  Calcul de la constante magique du second degré de l'ordre passé en paramètre.
    k2 = pOrdre * (pOrdre * pOrdre + 1) * ( 2 * pOrdre * pOrdre + 1) / 6;

    //  Calcul de la somme des éléments du vecteur au carré.
    sommeVecteur = 0;
    for  (i = 1; i <= pOrdre; i++)  sommeVecteur += pVecteur[i] * pVecteur[i];

    //  Mise à jour de la valeur de retour.
    if (sommeVecteur == k2)
        retour = 1;
    else
        retour = 0;

    return retour;
}
//  _____________________________________________________________________________________


//  ---------------------------------------------------------------------------------------------------------------------------
//  isTroisiemeOrdre
//  Fonction permettant de vérifier si une ligne magique possède également la magie
//      du troisième ordre.
//  Paramètres :
//      - pVecteur : Ligne magique à tester
//      - pOrdre : ordre de grandeur de la ligne à vérifier.
//  Valeur retournée : La fonction retourne 1 si la ligne possède également la magie du
//      troisième ordre ou 0 si elle est simplement magique.
//  ---------------------------------------------------------------------------------------------------------------------------
int isTroisiemeOrdre (int pVecteur[], int pOrdre)
{
    //  Taille du vecteur : minimum ordre + 1
    //  Element du vecteur dont on tient compte : 1 -> ordre
    int i, retour;
    double k3, sommeVecteur;

    //  Calcul de la constante magique du second degré de l'ordre passé en paramètre.
    k3 = pow (pOrdre * pOrdre * (pOrdre * pOrdre + 1) / 2, 2);
    k3 /= pOrdre;

    //  Calcul de la somme des éléments du vecteur au carré.
    sommeVecteur = 0;
    for  (i = 1; i <= pOrdre; i++)  sommeVecteur += pow (pVecteur[i], 3);

    //  Mise à jour de la valeur de retour.
    if (sommeVecteur == k3)
        retour = 1;
    else
        retour = 0;

    return retour;
}
//  _____________________________________________________________________________________
```

A.3 Contenu du CD-Rom joint à ce travail

Le CD-Rom fourni en annexe contient les programmes écrits pour ce travail ainsi que quelques fichiers Excel contenant des carrés générés qu'il était plus pratique de fournir sous ce format.

Le répertoire « Exemples de carrés générés »

Ce répertoire reprend une partie des carrés générés à l'aide de l'algorithme développé, et ce, pour les ordres de grandeurs de 5 à 23.

Le répertoire « Génération de carrés »

Ce répertoire reprend les programmes suivant :

- ***GenerationCarreXbase1*** : Ils permettent de générer des carrés d'ordre X en base 1 en utilisant une interface graphique permettant d'encoder les diagonales de manière conviviale et de sauvegarder les carrés générés dans un fichier « dataX.txt » Les interfaces ont été développées pour les ordres X compris entre 5 et 11.
- ***GenerationCarreXbase0*** : Ils permettent de générer des carrés d'ordre X en base 0 en utilisant une interface graphique permettant d'encoder les diagonales de manière conviviale et de sauvegarder les carrés générés dans un fichier « dataX.txt » Les interfaces ont été développées pour les ordres X compris entre 5 et 11.
- ***GenerationOrdreX*** : Ce programme permet de générer des carrés de n'importe quel ordre compris entre 5 et 23. Il ne travaille qu'en base 1 et ne possède pas d'interface graphique. Ce programme peut théoriquement travailler avec n'importe quel ordre de grandeur mais il a été limité à 23 à cause du dimensionnement de la taille des tableaux et des paramètres de la recherche. Les carrés générés par ce programme sont sauvés dans un fichier « data.txt »
- ***GenerationLignesMagiques*** : Ce programme permet de générer dans un fichier « data.txt » les lignes magiques d'un ordre donné. Il fonctionne pour n'importe quel ordre inférieur à 15 mais devient extrêmement lent à partir de l'ordre 7.